الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 2

Digital Systems and Binary Numbers

Weekly Learning Outcomes

1. Explain Digital Systems/Design
2. Learn the number systems used in digital computers (Binary, Octal, and Hexadecimal) and how to convert between them
3. Learn how to perform arithmetic operations on binary numbers
4. Learn how data is represented in digital system

Required Reading

1. Chapter 1 (Sections 1.1- 1.7) (Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapter 1 (pp 1-4), Chapter 2, Chapter 8 (John Seiffertt Digital Logic for Computing)

Computers are everywhere

- Digital circuits/processors are found in many forms and many places

Digital scale

Personal Computers

Gaming consoles

Cell phones

Healthcare equipment

https://commons.wikimedia.org/wiki/Category:Digital_electronic_devices

Digital circuits / Systems

- Information can be stored and manipulated in analog and digital forms (more on this later)
- In many cases a system takes an input, process it and produce output

Input data(information) $\Rightarrow$ Processing $\Rightarrow$ output data (information)

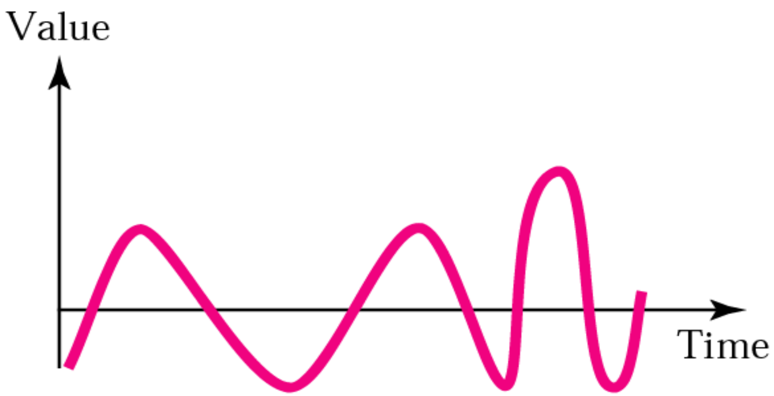
- System design involves:
  - Analyzing the problem that the system is trying to solve
  - Selecting the basic building blocks and design principals that help in solving the problem
  - Building a schematic of the design
  - Testing the design
- Digital systems works on information represented as digital signals

Signals

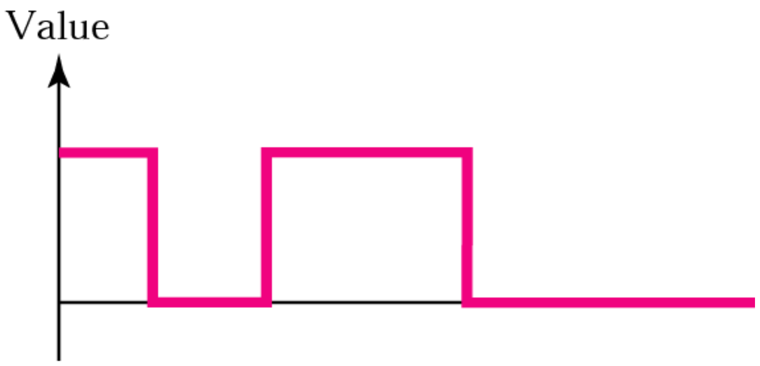
- Signals carries information that we are interested in consuming or manipulating.
- A signal is a physical or virtual quantity that varies with time (like sound), or space (like images), or another independent variable.
- Signals can be analog or digital

Signals

- ***Analog signals*** *can have an infinite number of values in a range*
- ***Digital signals*** *can have only a limited number of values*

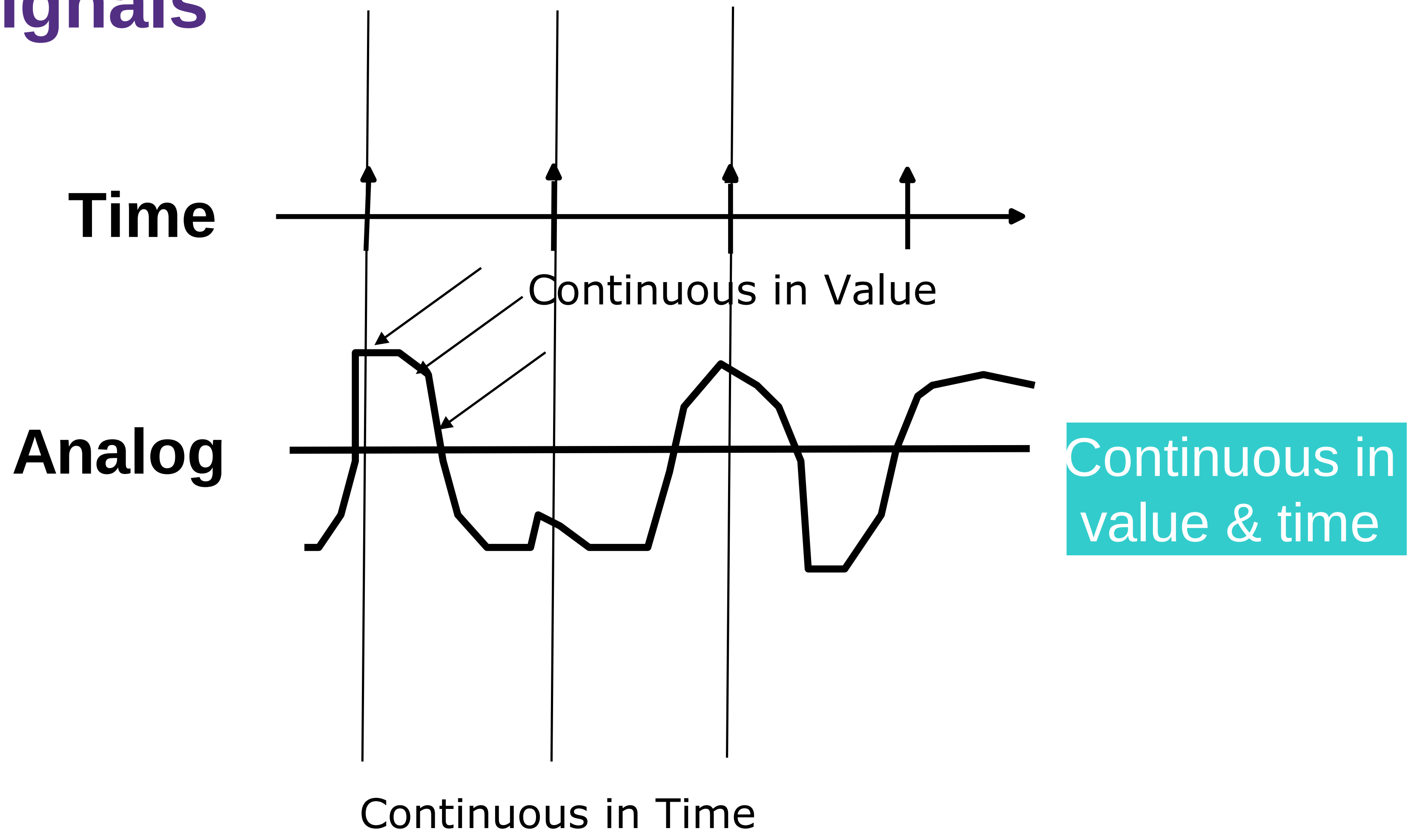


a. Analog signal



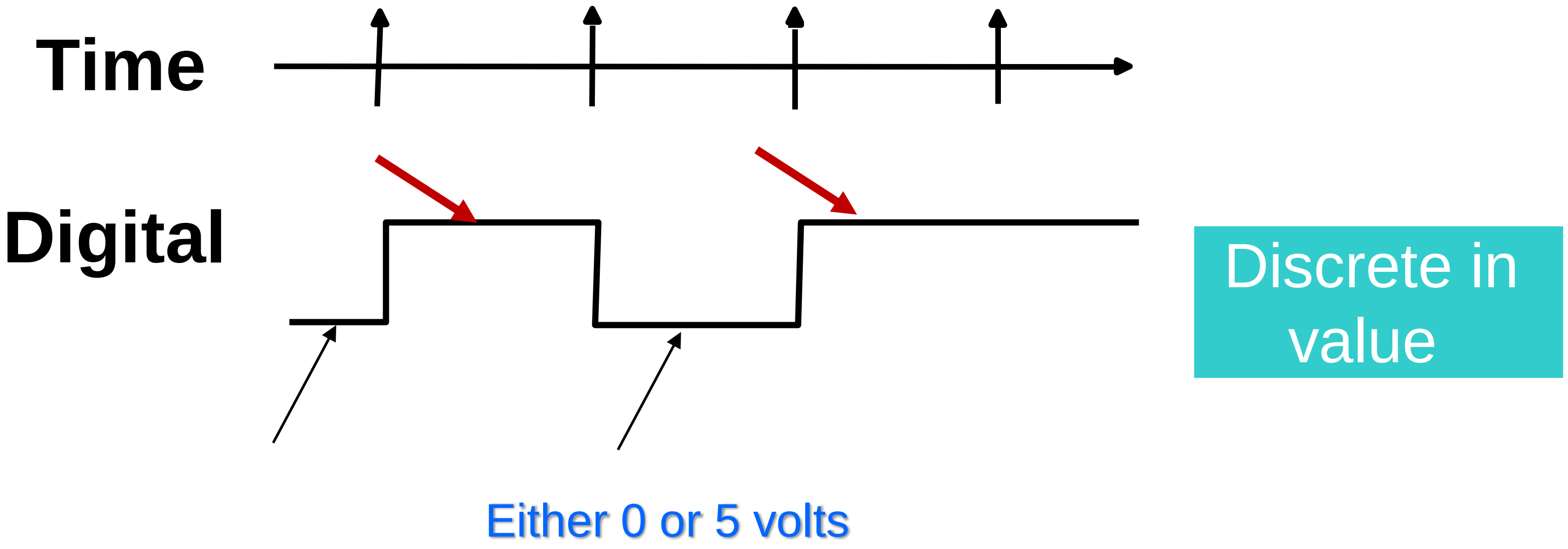
b. Digital signal

Analog Signals



Digital Signals

- For digital systems, the variable takes on discrete values (i.e., not continuous)
- Digital (Binary) values can be represented by:
  - digits 0 and 1
  - False (F) and True (T)
  - +5 Volt and 0 Volt



- Number Systems and numbers representation

Number Systems – Representation

- A number with radix r is represented by a string of digits:
  $A_{n-1}A_{n-2} \ldots A_1A_0 \,.\, A_{-1}\, A_{-2} \ldots A_{-m+1}\, A_{-m}$
  in which $0 \le A_i < r$ and "." is the radix point.
- The string of digits represents the power series:

$$(\text{Number})_r = \left(\sum_{i=0}^{i=n-1} A_i \cdot r^i\right) + \left(\sum_{j=-m}^{j=-1} A_j \cdot r^j\right)$$

(Integer Portion) + (Fraction Portion)

Decimal Number System

- Base (also called radix) = 10
  - 10 digits { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- Digit Position
  - Integer & fraction
- Formal Notation
- Digit Weight
  - Weight = (Base) Position
- Magnitude
  - Sum of “Digit x Weight”

2	1	0		-1	-2
5	1	2		7	4

$(512.74)_{10}$

100	10	1		0.1	0.01
500	10	2	●	0.7	0.04

$d_2 * B^2 + d_1 * B^1 + d_0 * B^0 + d_{-1} * B^{-1} + d_{-2} * B^{-2}$

Octal Number System

- Base = 8
    - 8 digits { 0, 1, 2, 3, 4, 5, 6, 7 }
- Weights
    - Weight = (Base) Position

64	8	1		1/8	1/64
5	1	2	●	7	4
2	1	0		-1	-2

- Formal Notation
- Magnitude
    - Sum of “Digit x Weight”

$$(512.74)_8$$

$$5 *8^{2}+1 *8^{1}+2 *8^{0}+7 *8^{-1}+4 *8^{-2}$$

$$=(330.9375)_{10}$$

Octal Number System: Example

- For Example,
  - $(27)_8$ can be expressed as: $(\quad)_{10}$
    - $(2 \times 8^1) + (7 \times 8^0) \rightarrow (2 \times 8) + (7 \times 1) \rightarrow 16 + 7 = (23)_{10}$
-
  - $(17.1)_8$ can be expressed as: $(\quad)_{10}$
    - $(1 \times 8^1) + (7 \times 8^0) + (1 \times 8^{-1}) \rightarrow 8 + 7 + 0.125 \rightarrow (15.125)_{10}$

Hexadecimal Number System

- Base = 16
  - 16 digits { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }

    A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- Weights
  - Weight = (Base) Position
- Formal Notation
- Magnitude
  - Sum of “Digit x Weight”

256	16	1		1/16	1/256
1	E	5	●	7	A
2	1	0		-1	-2

$(1E5.7A)_{16}$

$1 * 16^{2} + 14 * 16^{1} + 5 * 16^{0} + 7 * 16^{-1} + 10 * 16^{-2}$

$= (485.4765625)_{10}$

Hex to Decimal

- Just multiply each hex digit by decimal value and add the results.

Position 2 1 0

$$(2\ a\ c)_{16}$$

$$2 \cdot 16^{2} + 10 \cdot 16^{1} + 12 \cdot 16^{0} = (684)_{10}$$

3 2 1 0	3 2 1 0	3 2 1 0	3 2 1 0
16^3	16^2	16^1	16^0
4096	256	16	1

Dec	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f

- Binary Numbers

Binary Number System

- Base = 2
  - 2 digits { 0, 1 }, called binary digits or “bits”
- Weights
  - Weight = (Base) Position
- Magnitude
  - Sum of “Bit x Weight”
- Formal Notation
- Groups of bits

4 bits = Nibble $\boxed{1\ 0\ 1\ 1}$

8 bits = Byte $\boxed{1\ 1\ 0\ 0\ 0\ 1\ 0\ 1}$

4	2	1		1/2	1/4
$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	●	$\boxed{0}$	$\boxed{1}$
2	1	0		-1	-2

$$1 * 2^{2} + 0 * 2^{1} + 1 * 2^{0} + 0 * 2^{-1} + 1 * 2^{-2}$$

$$= (5.25)_{10}$$

$$(101.01)_{2}$$

Binary → Decimal: Example

7	6	5	4	3	2	1	0	position
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	value
128	64	32	16	8	4	2	1	

What is $(10011100)_2$ in decimal?

7	6	5	4	3	2	1	0	position
1	0	0	1	1	1	0	0	Binary #

$$128 + 0 + 0 + 16 + 8 + 4 + 0 + 0 = (156)_{10}$$

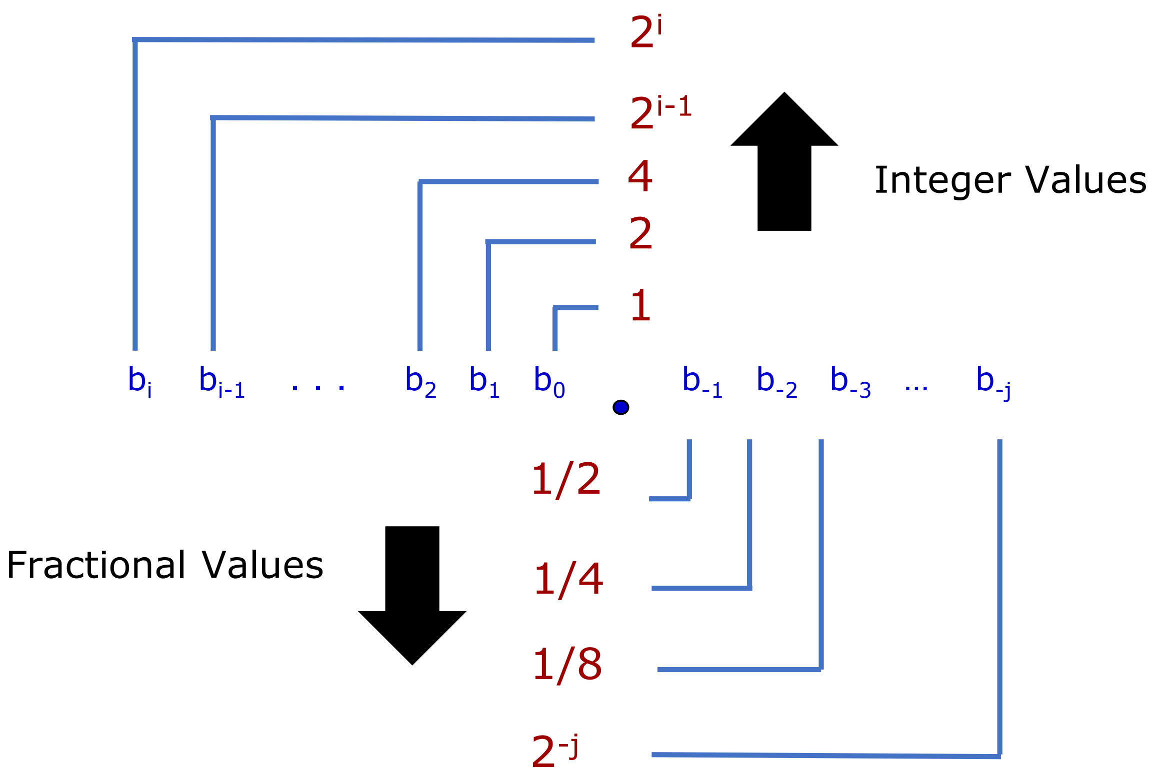
Binary Numbers

- Examples:
  - (00)2 → (0)10
  - (01)2 → (1)10
  - (0000001)2 → (1)10
  - (10)2 → (2)10
  - (010)2 → (2)10
  - (11)2 → (3)10
  - (100)2 → (4)10
  - (1001010101000)2

- Strings of binary digits (“bits”)
  - One bit can store a number from 0 to 1
  - n bits can store numbers from 0 to 2n-1

Binary Fractions

- Like dec, oct, and hex



decimal number = $\sum_{k=-j}^{i} b_k \, 2^k$

Example 1: Binary to Decimal

Position	2	1	0		-1	-2
	$(1$	0	1	$.$	1	$1)_2$
	1×2^2 +	0×2^1 +	$1\text{x}2^0$	+	$1\text{x}2^{-1}$ +	$1\text{x}2^{-2}$

$(5 \text{ and } ¾)_{10}$

Example 2: Binary to Decimal

Position	0	-1	-2	-3	-4	-5	-6
	(0 .	1	1	1	1	1	$1)_2$
	0×2^0	$+ \; 1 \times 2^{-1}$	$+ \; 1\text{x}2^{-2}$	$+ \; 1\text{x}2^{-3}$	$+ \; 1\text{x}2^{-4}$	$+ \; 1\text{x}2^{-5}$	$+ \; 1\text{x}2^{-6}$

$(63/64)_{10}$

Note: (1) Numbers of the form $0.11111\ldots_2$ are just below 1.0

(2) Short form notation for such numbers is $1.0 - \varepsilon$

- Base conversion

Conversion Between Bases

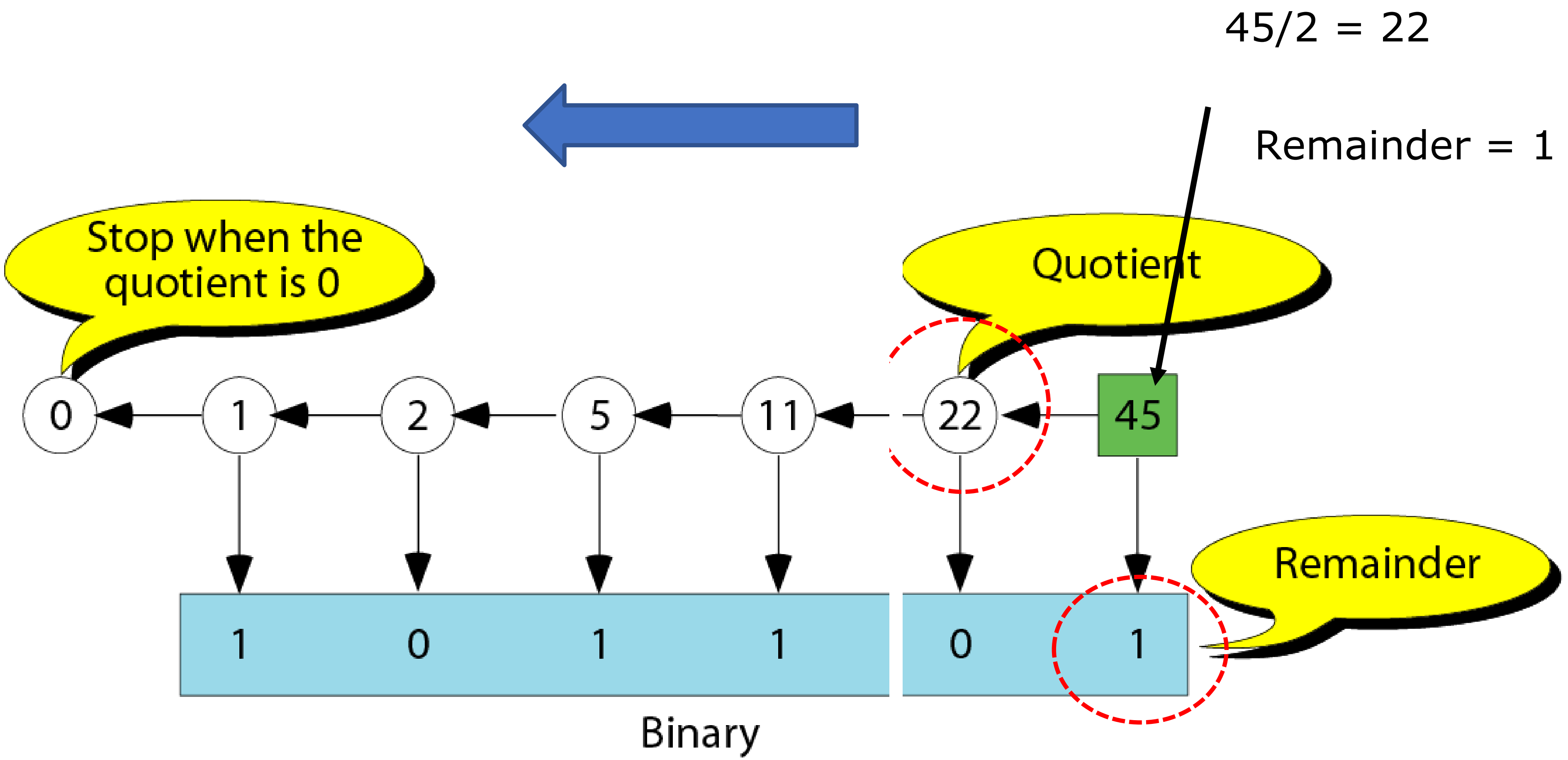
To convert from one base to another:

1) **Convert the Integer Part**

2) **Convert the Fraction Part**

3) **Join the two results with a radix point**

Decimal to Binary Conversion



Divide the Decimal number by the base (i.e. for binary divide by 2) .1
The remainder is used to form the LSB of the binary number .2
The quotient will be divided again by 2 until the quotient reaches 0 .3

Decimal (Integer) to Binary Conversion

- Divide the number by the ‘Base’ (=2)
- Take the remainder (either 0 or 1) as a coefficient
- Take the quotient and repeat the division

Example: $(13)_{10}$

		Quotient	Remainder	Coefficient	
	$13 / 2 =$	6	1	$a_0 = 1$	LSB
	$6 / 2 =$	3	0	$a_1 = 0$	
	$3 / 2 =$	1	1	$a_2 = 1$	
STOP!	$1 / 2 =$	0	1	$a_3 = 1$	MSB

Answer: $(13)_{10} = (a_3\, a_2\, a_1\, a_0)_2 = (1101)_2$

MSB **LSB**

Decimal (Fraction) to Binary Conversion

- Multiply the number by the ‘Base’ (=2)
- Take the integer (either 0 or 1) as a coefficient
- Take the resultant fraction and repeat multiplication

Example: $(0.625)_{10}$

		Integer		Fraction	Coefficient	
	0.625 * 2 =	1	.	25	$a_{-1} = 1$	MSB
	0.25 * 2 =	0	.	5	$a_{-2} = 0$	
STOP!	0.5 * 2 =	1	.	0	$a_{-3} = 1$	LSB

Answer: $(0.625)_{10} = (0.a_{-1}\, a_{-2}\, a_{-3})_2 = (0.101)_2$

MSB LSB

Decimal to Octal Conversion

Example: $(175)_{10}$

	Quotient	Remainder	Coefficient	
175 / 8 =	21	7	$a_0 = 7$	MSB
21 / 8 =	2	5	$a_1 = 5$	
STOP! 2 / 8 =	0	2	$a_2 = 2$	LSB

Answer: $(175)_{10} = (a_2\, a_1\, a_0)_8 = (257)_8$

Example: $(0.3125)_{10}$

	Integer		Fraction	Coefficient
0.3125 * 8 =	2	.	5	$a_{-1} = 2$
STOP! 0.5 * 8 =	4	.	0	$a_{-2} = 4$

Answer: $(0.3125)_{10} = (0.a_{-1}\, a_{-2}\, a_{-3})_8 = (0.24)_8$

Decimal to Hex

$(684)_{10}$

684/16 = 42 rem 12 — c

42/16 = 2 rem 10 — ac

2/16 = 0 rem 2 — 2ac

❸❷❶⓿	❸❷❶⓿	❸❷❶⓿	❸❷❶⓿
16^3	16^2	16^1	16^0
4096	256	16	1

Dec	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f

Hexadecimal (Base 16)

- Strings of 0's and 1's too hard to write
- Use base-16 or hexadecimal – 4 bits

2nd LSB changes every two clock cycles

LSB changes every clock cycle

Dec	Bin	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Dec	Bin	Hex
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

Why use base 16?

- **Power of 2**
- **Size of byte**

Hex to Binary

- Convention – write 0x (prefix)
- before number
- Hex to Binary – just convert digits

0x2ac $(2ac)_{16}$

0010 1010 1100

$0x2ac = (001010101100)_2$

No magic – remember hex digit = 4 bits

Bin	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	a
1011	b
1100	c
1101	d
1110	e
1111	f

Octal to Binary

- 3-bits required for every Octal number
- Octal to Binary – just convert digits

$(273)_8$

010 111 011

$$\boxed{(273)_8 = (010111011)_2}$$

Bin	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

No magic – remember octal digit = 3 bits

Binary – Hexadecimal Conversion

- 16 = 24
- Each group of 4 bits represents a hexadecimal digit

Example:

Pad with Zeros

0001 0100

$(1\,0\,1\,1\,0\,.\,0\,1)_2$

$(1 \quad 6 \quad . \quad 4)_{16}$

Hex	Binary
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
A	1 0 1 0
B	1 0 1 1
C	1 1 0 0
D	1 1 0 1
E	1 1 1 0
F	1 1 1 1

Works both ways (*Binary* to *Hex* & *Hex* to *Binary*)

Binary to Hex

- Just convert groups of 4 bits

$$(\cancel{101001101111011})_2$$

$$0101 \mid 0011 \mid 0111 \mid 1011$$

$$\downarrow \qquad \downarrow \qquad \downarrow \qquad \downarrow$$

$$5 \qquad 3 \qquad 7 \qquad b$$

$$(0101001101111011)_2 = \text{0x537b} = (537b)_{16}$$

Bin	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	a
1011	b
1100	c
1101	d
1110	e
1111	f

Octal – Hexadecimal Conversion

- Convert to Binary as an intermediate step

Example:

$$(\underbrace{\overbrace{0\,1\,0}^{2}}_{}\ \underbrace{\overbrace{1\,1\,0}^{6}}_{} \,.\, \underbrace{\overbrace{0\,1\,0}^{2}}_{})_2$$

$(2 \quad 6 \,.\, 2)_8$

Pad Zeros

$(0\,1\,0\,1\,1\,0\,.\,0\,1\,0)_2$

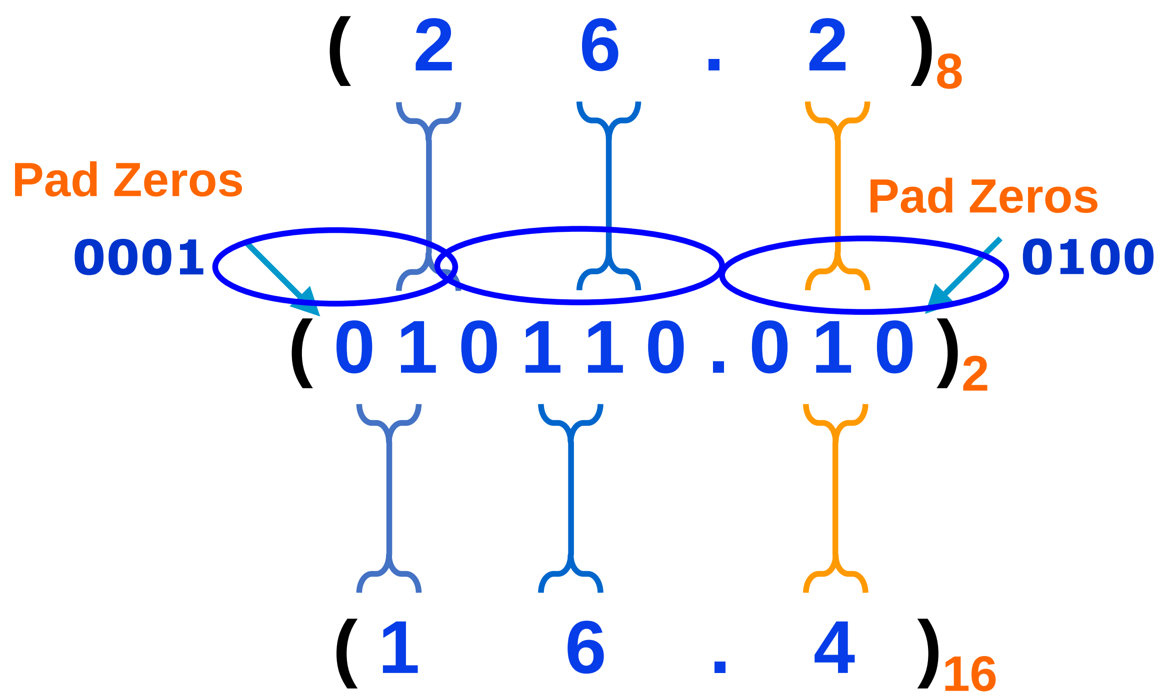
Pad Zeros

Bin	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Octal – Hexadecimal Conversion

- Convert to Binary as an intermediate step

Example:



Works both ways (*Octal* to *Hex* & *Hex* to *Octal*)

Bin	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	a
1011	b
1100	c
1101	d
1110	e
1111	f

- Binary Arithmetic

Addition

- Decimal Addition

$$\begin{array}{r} & 1 & 1 & \\ & & 5 & 5 \\ + & & 5 & 5 \\ \hline & 1 & 1 & 0 \end{array}$$

← Carry

= *Ten* ≥ *Base*

➔ **Subtract a Base**

Binary Addition

- Adding bits:
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = (1) 0 carry 2 in Binary
  - 1 + 1 + 1 = (1) 1 carry 3 in Binary
- Adding integers:

					1	1	0		
	0	0	0	0	1	1	1	$(1)_2$ =	$(7)_{10}$
+	0	0	0	0	1	1	0	$(0)_2$ =	$(6)_{10}$
=	0	0	0	1	(1)1	(1)0	(0)	$(1)_2$ =	$(13)_{10}$

Binary Addition

- Column Addition

$$
\begin{array}{cccccccc}
 & 1 & 1 & 1 & 1 & 1 & 1 & \\
 & & 1 & 1 & 1 & 1 & 0 & 1 \\
+ & & & 1 & 0 & 1 & 1 & 1 \\
\hline
 & 1 & 0 & 1 & 0 & 1 & 0 & 0
\end{array}
$$

Verify the result

$= (61)_{10}$

$= (23)_{10}$

$= (84)_{10}$

$\geq (2)_{10}$

★ **Recall:**

- ♦ **1 + 1 = 1 0**
- ♦ **1 + 1 + 1 = 1 1**

Complements of numbers

- They are used to simplify the subtraction operation
- They can be formed in any base, we will focus on binary numbers system
- We have 1’s complement and 2’s complement
- For 1’s complement change all 0 to 1 and all 1 to 0
- For 2’s complement obtain 1’s complement and add 1

1’s complement example

1011000

0 1 0 0 1 1 1

2's complement

	1101100
Carry	000011
Take 1's complement	0010011 +
Add 1	1
	0010100

Subtraction with Complements

- To subtract N from M (M-N) you need to
  - Take 2’s complement of N
  - Add M to the 2’s complement of N
  - If M ≥ N a carry will be produced at the end which can be discarded
  - If M < N the result will be negative. The result will be the 2’s complement of (N-M) to get it to familiar form take the 2’s complement of the result and add minus sign

Subtraction example

- Given X = 1010100 and Y = 1000011 find X-Y and Y-X
- First let us find X - Y

```
                          1111100
                   X =  1010100
2's complement of Y =  0111101+
                       --------
                       10010001
```

There is a carry
Which means the result is positive
Just discard the carry

→ X – Y = 0010001

Getting 2's complement of Y

	1000011
1's complement	0111100
Adding 1	1+

	0111101

Check the answer

X = 84, Y = 67

X-Y = 84 – 67 = 17

0010001 =

$1*2^4 + 1*2^0$ =

16+1 = 17

Subtraction example continued

- Given X = 1010100 and Y = 1000011 find X-Y and Y-X
- Now let us find Y-X

0000000

Y = 1000011

2’s complement of Y = 0101100+

1101111

There is no carry
Which means the result is negative
Take the 2’s complement and add (-) sign

➔ Y-X = -(2’s complement of 1101111) = - (0010001)

Getting 2’s complement of X

	1010100
1’s complement	0101011
Adding 1	1+

	0101100

Check the answer

X = 84, Y = 67

Y-X = 67– 84 = -17

- (0010001) =

$-(1*2^4 + 1*2^0) =$

-(16+1) = -17

- Binary Codes

Binary codes

- Digital system signals have two distinct values only
- Information in real world need more than two values to be represented more accurately
- An n-bits binary code is a group of n bits that can have 2^n distinct values
- Numerical values can be represented by directly binary numbers using the conversion techniques we discussed in the class
- Non-Numerical values, such as alphabet characters, needs a mapping between distinct bit-group values and the non-numerical values

Non-numeric Binary Codes

- Given *n* binary digits (called bits), a binary code is a mapping from a set of represented elements **to** a subset of the 2^n binary numbers.
- Example: A binary code for the seven colors of the rainbow
- Code 100 is not used

Color	Binary Number
Red	000
Orange	001
Yellow	010
Green	011
Blue	101
Indigo	110
Violet	111

Number of Bits Required

- In the previous example (rainbow colors) we used 3 bits and one of the values is not used, can we use less bits for representation?
  - The minimum number of bits depends on the number of elements to be represented.
- Given M elements to be represented by a binary code, the minimum number of bits, n, needed, satisfies the following relationships:
  - $2^n \geq M > 2^{(n-1)}$
  - $n = \lceil \log2\ M \rceil$ where $\lceil x \rceil$, is called the ceiling function, i.e., the integer greater than or equal to x.

Number of Bits Required (examples)

- Example: How many bits are required to represent decimal digits with a binary code?
  - M = 10, hence n = ceiling ($\log_2 10$) = ceiling (3.3219) = 4
  - Checking: $2^4 = 16 \geq 10 > 2^3 = 8$
- Example: Given a machine with ***[off, low power, medium power, high power, ultra high power]*** states how many bits are required to represent these states?
  - M = 5, hence n = ceiling ($\log_2 5$) = ceiling (2.32192) = 3
  - Checking $2^3 = 8 \geq 5 > 2^2 = 4$

Binary Codes

- Group of n bits
    - Up to 2n combinations
    - Each combination represents an element of information
- Binary Coded Decimal (BCD)
    - Each Decimal Digit is represented by 4 bits
    - (0 – 9) ⇨ Valid combinations
    - (10 – 15) ⇨ Invalid combinations

Decimal	BCD
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Gray Code

- Only one bit changes from one code to the next code
- Less power consumed since less transistors go on and off.
- Different than Binary

Decimal	Gray
00	0000
01	0001
02	0011
03	0010
04	0110
05	0111
06	0101
07	0100
08	1100
09	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

Binary
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Conversion or Coding?

- **Do NOT mix up conversion of a decimal number to a binary number with coding a decimal number with a BINARY CODE.**

- **$(13)_{10} = (1101)_2$ (This is conversion)**

- **$(13)_{BCD} \Leftrightarrow (0001|0011)_{BCD}$ (This is coding)**

Advantages/Disadvantages?

BCD: Advantages/Disadvantages

- Disadvantage:
  - It is obvious that a BCD number needs more bits than its equivalent binary value
    - $(26)_{10} = (11010)_2$
    - $(26)_{10} = (0010\ 0110)_{BCD}$
- Advantages:
  - Computer input/output data are handled by people who use the decimal system. So it is easier to convert back/forth to BCD.

ASCII Code

American Standard Code for Information Interchange

- Each letter or symbol is assigned a number from 0 to 127.
- For example the ASCII code for uppercase A is ‘1000001’ which is 65 decimal.

Info	7-bit Code
A	1000001
B	1000010
⋮	⋮
Z	1011010
a	1100001
b	1100010
⋮	⋮
z	1111010
@	1000000
?	0111111
+	0101011

American Standard Code for Information Interchange (ASCII)

	$B_7B_6B_5$							
$B_4B_3B_2B_1$	000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 3

Boolean Algebra and Logic Gates

CONTENTS

- Binary Storage and Registers
- Binary Logic
- Axiomatic Definition of Boolean Algebra
- Basic Theorems and Properties of Boolean Algebra

Weekly Learning Outcomes

1. Understand Binary Storage and Binary Logic
2. Gain a basic understanding of postulates used to form algebraic structures.
3. Understand the basic theorems and postulates of Boolean algebra
4. Know how to apply DeMorgan’s theorems

Required Reading

1. Chapter 1 (1.8 to 1.9)
2. Chapter 2 (Sections 2.1- 2.4)

(Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

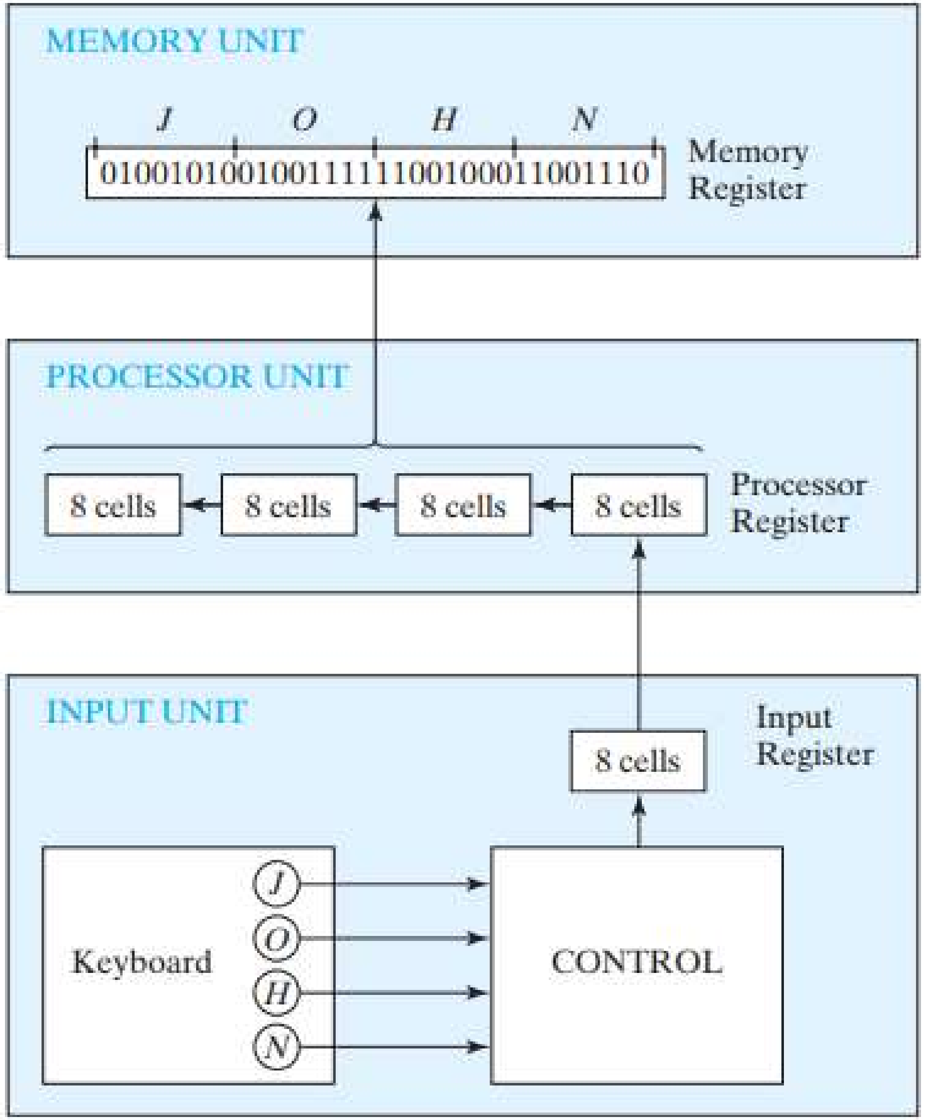
1. Chapter 2 (Digital Logic for Computing) (John Seiffertt Digital Logic for Computing)

Binary Storage and Registers

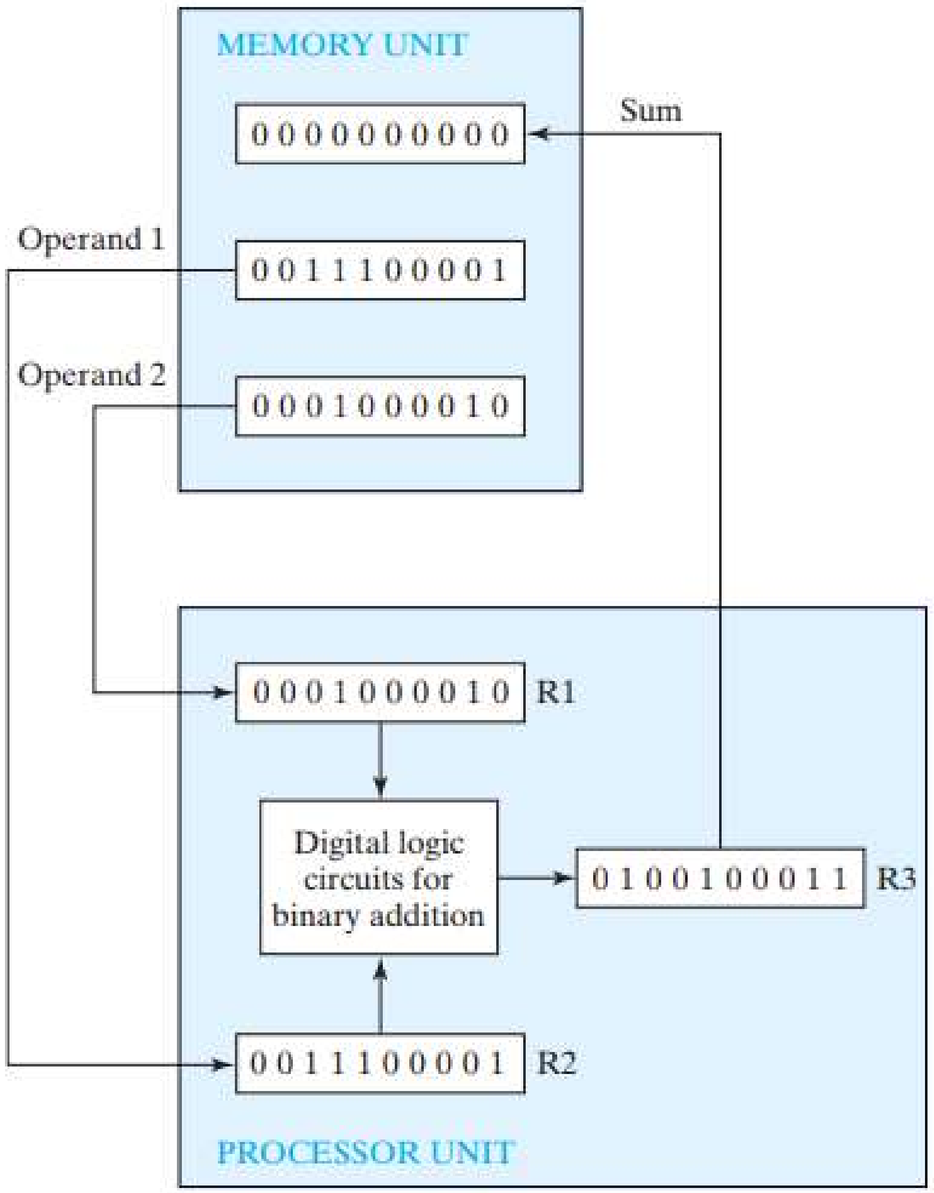
Registers

- A register is a group of binary cells. A register with n cells can store any discrete quantity of information that contains n bits.
- The state of a register is an n-tuple of 1’s and 0’s, with each bit designating the state of one cell in the register.
  for example,
  A 16-bit register with the following binary content:
  1100001111001001
- A register with 16 cells can be in one of 216 possible states the register can store any binary number from 0 to 216 – 1.

Transfer of Information among Registers



Binary Transformation Processing



Binary Logic

Binary Logic

- Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning.
- The two values the variables assume may be called by different names (*true* and *false, yes* and *no*, etc.), but for our purpose, it is convenient to think in terms of bits and assign the values 1 and 0.

Binary Logic

There are three basic logical operations: AND, OR, and NOT. Each operation produces a binary result, denoted by z.

1. AND:

This operation is represented by a dot or by the absence of an operator. For example, $x \cdot y = z$ or $xy = z$ is read "x AND y is equal to z." The logical operation AND is interpreted to mean that $z = 1$ if and only if $x = 1$ and $y = 1$; otherwise $z = 0$. (Remember that x, y, and z are binary variables and can be equal either to 1 or 0, and nothing else.) The result of the operation $x \cdot y$ is z

Binary Logic

2. OR:

This operation is represented by a plus sign. For example, $x + y = z$ is read "*x* OR *y* is equal to *z*," meaning that $z = 1$ if $x = 1$ or if $y = 1$ or if both $x = 1$ and $y = 1$. If both $x = 0$ and $y = 0$, then $z = 0$.

Binary Logic

3. NOT:

This operation is represented by a prime (sometimes by an overbar). For example, $x' = z$ (or $\bar{x} = z$) is read "not x is equal to z," meaning that z is what x is not. In other words, if $x = 1$, then $z = 0$, but if $x = 0$, then $z = 1$. The NOT operation is also referred to as the complement operation, since it changes a 1 to 0 and a 0 to 1, i.e., the result of complementing 1 is 0, and vice versa.

Truth Table

Truth Tables of Logical Operations

AND

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

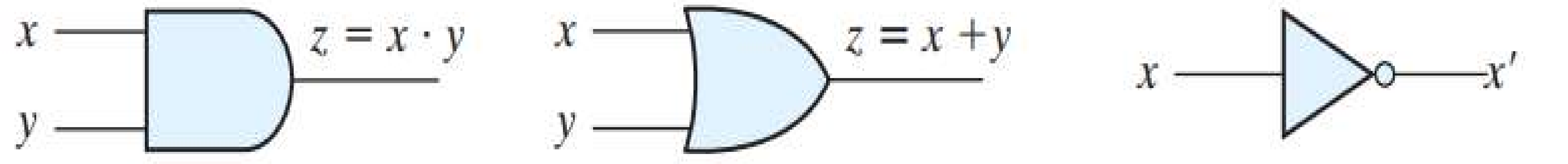
x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

x	x'
0	1
1	0

Logic Gates

Logic gates are electronic circuits that operate on one or more input signals to produce an output signal.



(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter

Boolean Algebra

Basic Definition

- Boolean algebra, like any other deductive mathematical system, may be defined with a set of elements, a set of operators, and a number of unproved axioms or postulates.
- A *set* of elements is any collection of objects, usually having a common property.

Common Postulates

1. Closure

- A set *S* is *closed* with respect to a binary operator if, for every pair of elements of *S*, the binary operator specifies a rule for obtaining a unique element of *S*.
- For example, the set of natural numbers $N = \{1, 2, 3, 4, \ldots\}$ is closed with respect to the binary operator + by the rules of arithmetic addition, since, for any $a, b \in N$, there is a unique $c \in N$ such that $a + b = c$.

Common Postulates

2. Associative law

A binary operator * on a set *S* is said to be *associative* whenever

$(x * y) * z = x * (y * z)$ for all $x, y, z \in S$

3. Commutative law

A binary operator * on a set *S* is said to be *commutative* whenever $x * y = y * x$ for all $x, y \in S$

Common Postulates

4. Identity element

- A set *S* is said to have an *identity* element with

  respect to a binary operation * on *S* if there exists an element $e \in S$ with the property that

  $e * x = x * e = x$ for every $x \in S$

- *Example:* The element 0 is an identity element with respect to the binary operator + on the

  set of integers $I = \{ \dots, -3, -2, -1, 0, 1, 2, 3, \dots \}$, since

  $x + 0 = 0 + x = x$ for any $x \in I$

  The set of natural numbers, *N*, has no identity element, since 0 is excluded from the set.

Common Postulates

5. Inverse

A set *S* having the identity element *e* with respect to a binary operator * is said to have an *inverse* whenever, for every $x \in S$, there exists an element $y \in S$ such that

$x * y = e$

Example: In the set of integers, *I*, and the operator + , with e = 0 , the inverse of an element *a* is (- a) , since a + (- a) = 0.

Common Postulates

6. Distributive law

If * and · are two binary operators on a set *S*, * is said to be *distributive* over · whenever
$x * (y \cdot z) = (x * y) \cdot (x * z)$

The operators and postulates have the following meanings:

The binary operator + defines addition.

The additive identity is 0

The additive inverse defines subtraction.

The binary operator · defines multiplication.

The multiplicative identity is 1.

For $a \neq 0$, the multiplicative inverse of $a = 1 / a$ defines division (i.e., $a \cdot 1 / a = 1$).

The only distributive law applicable is that of · over + :
$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

AXIOMATIC DEFINITION OF BOOLEAN ALGEBRA

common postulates

- In 1854, George Boole developed an algebraic system now called *Boolean algebra*.

- For the formal definition of Boolean algebra, we shall employ the postulates formulated by E. V. Huntington in 1904.

- Boolean algebra is an algebraic structure defined by a set of elements, *B*, together with two binary operators, + and · , provided that the following (Huntington) postulates are satisfied:

Huntington Postulates

- 1.
  1. The structure is closed with respect to the operator + .
  2. The structure is closed with respect to the operator · .

- 2.
  1. The element 0 is an identity element with respect to + ; that is, $x + 0 = 0 + x = x$.
  2. The element 1 is an identity element with respect to · ; that is, $x \cdot 1 = 1 \cdot x = x$.

Huntington Postulates

- 3.
  1. The structure is commutative with respect to + ; that is, $x + y = y + x$
  2. The structure is commutative with respect to · ; that is, $x \cdot y = y \cdot x$

- 4.
  1. The operator · is distributive over + ; that is, $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
  2. The operator + is distributive over · ; that is, $x + (y \cdot z) = (x + y) \cdot (x + z)$

Huntington Postulates

- 5. For every element $x \in B$, there exists an element $x' \in B$ (called the *complement* of *x*) such that (a) $x + x' = 1$ and (b) $x \cdot x' = 0$.

- 6. There exist at least two elements $x , y \in B$ such that $x \neq y$.

Comparing Boolean algebra with arithmetic and ordinary algebra (the field of real numbers), we note the following differences:

1. Huntington postulates do not include the associative law. However, this law holds for Boolean algebra and can be derived (for both operators) from the other postulates.

2. The distributive law of + over · (i.e., $x + (y \cdot z) = (x + y) \cdot (x + z)$) is valid for Boolean algebra, but not for ordinary algebra.

3. Boolean algebra does not have additive or multiplicative inverses; therefore, there are no subtraction or division operations.

- 4. Postulate 5 defines an operator called the *complement* that is not available in ordinary algebra.
- 5. Ordinary algebra deals with the real numbers, which constitute an infinite set of elements. Boolean algebra deals with the as yet undefined set of elements, *B*, but in the two-valued Boolean algebra defined next (and of interest in our subsequent use of that algebra), *B* is defined as a set with only two elements, 0 and 1.

Two-Valued Boolean Algebra

Two-Valued Boolean Algebra

- A two-valued Boolean algebra is defined on a set of two elements, B = { 0, 1 } , with rules for the two binary operators + and ·

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

These rules are exactly the same as the AND, OR, and NOT operations

Two-Valued Boolean Algebra

We now show that the Huntington postulates are valid for the set

$B = \{0, 1\}$ and the two binary operators + and .

1. That the structure is *closed* with respect to the two operators is obvious from the tables, since the result of each operation is either 1 or 0 and $1, 0 \in B$.
2. From the tables, we see that
   (a) $0 + 0 = 0$ $\quad\quad$ $0 + 1 = 1 + 0 = 1$;
   (b) $1 \cdot 1 = 1$ $\quad\quad$ $1 \cdot 0 = 0 \cdot 1 = 0$.
   This establishes the two *identity elements,* 0 for + and 1 for . , as defined by postulate 2.
3. The *commutative* laws are obvious from the symmetry of the binary operator tables.

Two-Valued Boolean Algebra

4. (a) The *distributive* law $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ can be shown to hold from the operator tables by forming a truth table of all possible values of x, y, and z. For each combination, we derive $x \cdot (y + z)$ and show that the value is the same as the value of $(x \cdot y) + (x \cdot z)$:

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

$y + z$	$x \cdot (y + z)$
0	0
1	0
1	0
1	0
0	0
1	1
1	1
1	1

$x \cdot y$	$x \cdot z$	$(x \cdot y) + (x \cdot z)$
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	1	1
1	0	1
1	1	1

(b) The *distributive* law of + over $\cdot$ can be shown to hold by means of a truth table similar to the one in part (a).

Two-Valued Boolean Algebra

5. From the complement table, it is easily shown that
(a) $x + x' = 1$, since $0 + 0' = 0 + 1 = 1$ and $1 + 1' = 1 + 0 = 1$
(b) $x \cdot X' = 0$, since $0 \cdot 0' = 0 \cdot 1 = 0$ and $1 \cdot 1' = 1 \cdot 0 = 0$
Thus, postulate 1 is verified.

6. Postulate 6 is satisfied because the two-valued Boolean algebra has two elements, 1 and 0, with $1 \neq 0$.

Basic Theorems and Properties of Boolean Algebra

Basic Theorems

Table 2.1
Postulates and Theorems of Boolean Algebra

Postulate 2	(a)	$x + 0 = x$	(b)	$x \cdot 1 = x$
Postulate 5	(a)	$x + x' = 1$	(b)	$x \cdot x' = 0$
Theorem 1	(a)	$x + x = x$	(b)	$x \cdot x = x$
Theorem 2	(a)	$x + 1 = 1$	(b)	$x \cdot 0 = 0$
Theorem 3, involution		$(x')' = x$		
Postulate 3, commutative	(a)	$x + y = y + x$	(b)	$xy = yx$
Theorem 4, associative	(a)	$x + (y + z) = (x + y) + z$	(b)	$x(yz) = (xy)z$
Postulate 4, distributive	(a)	$x(y + z) = xy + xz$	(b)	$x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a)	$(x + y)' = x'y'$	(b)	$(xy)' = x' + y'$
Theorem 6, absorption	(a)	$x + xy = x$	(b)	$x(x + y) = x$

Basic Theorems

The postulates are basic axioms of the algebraic structure and

need no proof. The theorems must be proven from the postulates.

Proofs of the theorems with one variable are presented next.

Basic Theorems

THEOREM 1(a): $x + x = x$.

Statement	Justification
$x + x = (x + x) \cdot 1$	postulate 2(b)
$= (x + x)(x + x')$	5(a)
$= x + xx'$	4(b)
$= x + 0$	5(b)
$= x$	2(a)

Basic Theorems

THEOREM 1(b): $x \cdot x = x$.

Statement	Justification
$x \cdot x = xx + 0$	postulate 2(a)
$= xx + xx'$	5(b)
$= x(x + x')$	4(a)
$= x \cdot 1$	5(a)
$= x$	2(b)

Basic Theorems

THEOREM 2(a): $x + 1 = 1.$

Statement	Justification
$x + 1 = 1 \cdot (x + 1)$	postulate 2(b)
$= (x + x')(x + 1)$	5(a)
$= x + x' \cdot 1$	4(b)
$= x + x'$	2(b)
$= 1$	5(a)

THEOREM 2(b): $x \cdot 0 = 0$ by duality.

Basic Theorems

THEOREM 3: $(x')' = x$. From postulate 5, we have $x + x' = 1$ and $x \cdot x' = 0$, which together define the complement of x. The complement of x' is x and is also $(x')'$.

Therefore, since the complement is unique, we have $(x')' = x$. The theorems involving two or three variables may be proven algebraically from the postulates and the theorems that have already been proven.

Basic Theorems

THEOREM 6(a): $x + xy = x$.

Statement	Justification
$x + xy = x \cdot 1 + xy$	postulate 2(b)
$= x(1 + y)$	4(a)
$= x(y + 1)$	3(a)
$= x \cdot 1$	2(a)
$= x$	2(b)

THEOREM 6(b): $x(x + y) = x$ by duality.

Basic Theorems

The theorems of Boolean algebra can be proven by means of truth tables. In truth tables, both sides of the relation are checked to see whether they yield identical results for all possible combinations of the variables involved. The following truth table verifies the first absorption theorem:

Basic Theorems

x	y
0	0
0	1
1	0
1	1

xy	$x + xy$
0	0
0	0
0	1
1	1

Operator Precedence

- The operator precedence for evaluating Boolean expressions is (1) parentheses, (2) NOT, (3) AND, and (4) OR. In other words, expressions inside parentheses must be evaluated before all other operations. The next operation that holds precedence is the complement, and then follows the AND and, finally, the OR.

Thank You

الجامعة السعودية الإلكترونية

SAUDI ELECTRONIC UNIVERSITY

2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 4

Boolean Algebra and Logic Gates

Weekly Learning Outcomes

1. Define Boolean functions/Boolean expression
2. Learn how to express a Boolean function in canonical and standard forms
3. Learn the logic gates symbols and their truth tables
4. Learn how to manipulate Boolean expression to reduce the required number of logic gates

Required Reading

1. Chapter 2 (Sections 2.5- 2.8) (Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapters 3 (John Seiffertt Digital Logic for Computing)
2. https://www.tutorialspoint.com/digital_circuits/digital_circuits_canonical_standard_forms.htm

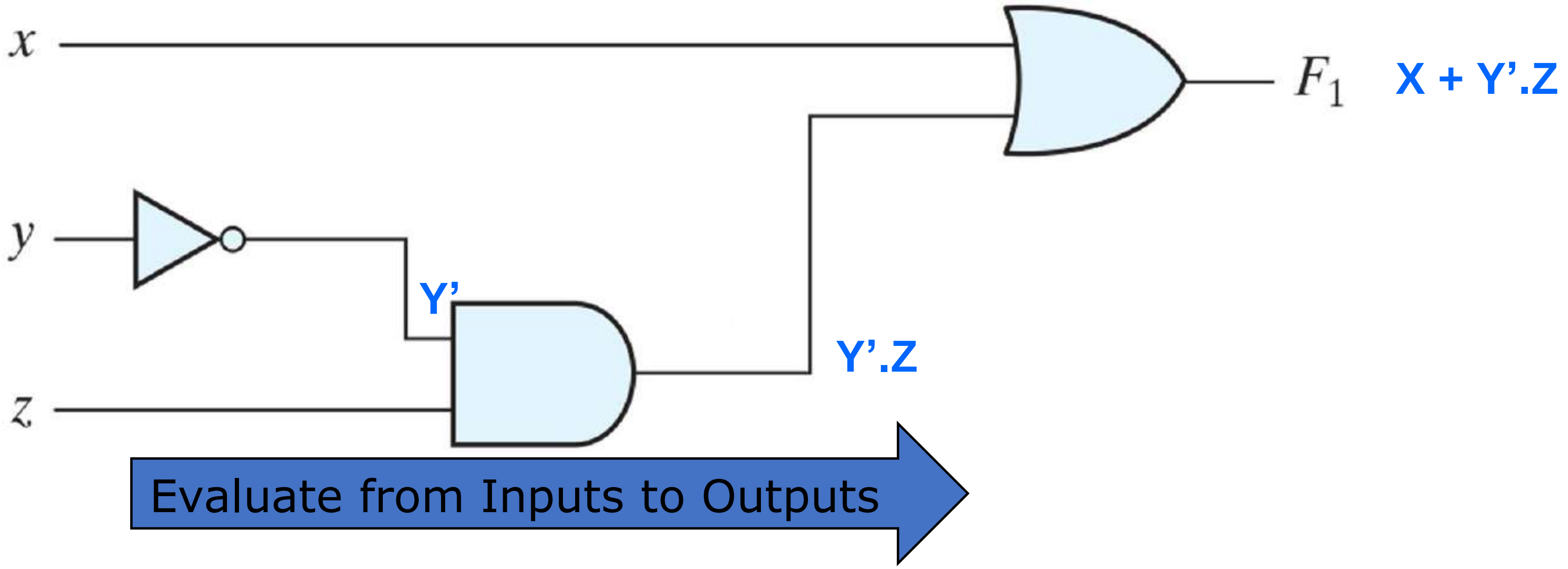
Boolean Functions

- An algebraic function is an evaluation of a set of variables and operations
  - F(x, y) = x + 2*y (variables F, x, y; operations +,*, =)
- Boolean algebraic functions use logic operations only
- Boolean variables values and function's values can be 1, or 0
- Example: $F_1 = x + y^{`} z$
  - Function: F_1
  - Variables: x, y, z
  - Boolean Expression: $x + y^{`} z$
  - Logical Operations:

$$x \overset{\text{OR}}{+} y^{\overset{\text{NOT}}{`}} \underset{\text{AND}}{\cdot} z$$

Logic Diagram → Boolean Expression

- Logic Diagram



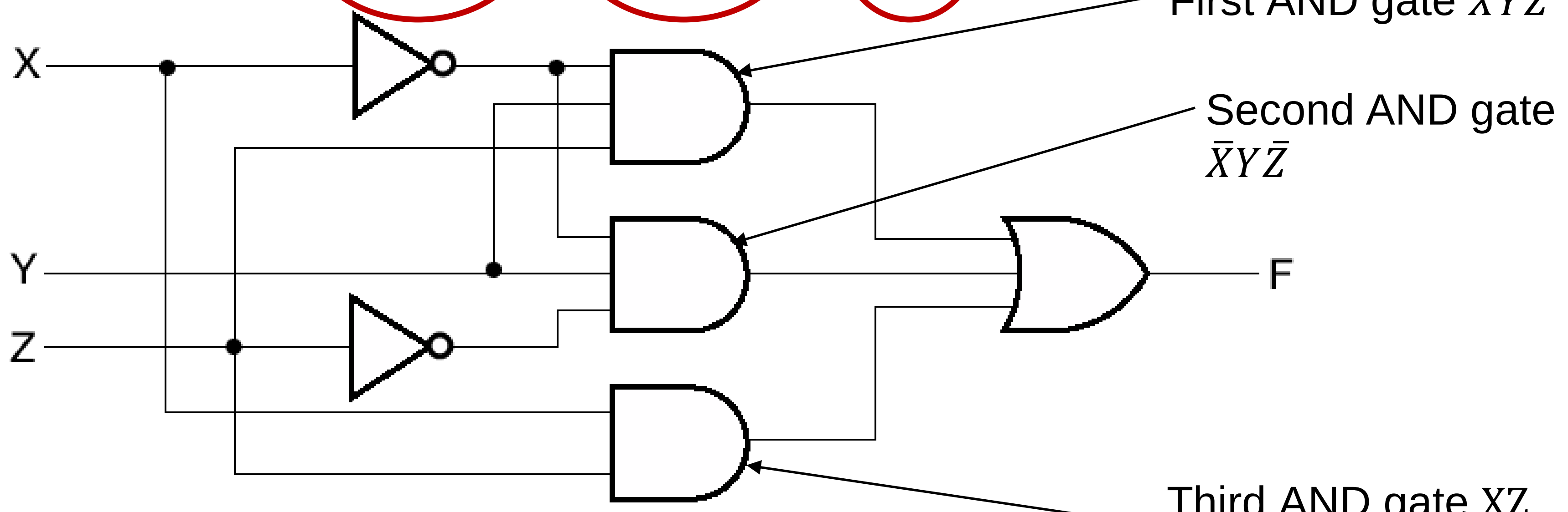
- What is the Boolean expression?

$$F = X + \overline{Y}.Z$$

Boolean Expression → Logic Diagram

- Consider function

$$F = \bar{X}YZ + \bar{X}Y\bar{Z} + XZ$$



(a) $F = \bar{X}YZ + \bar{X}Y\bar{Z} + XZ$

How many gates do we need to implement this function?

Truth Tables

- A truth table spells all input combinations and their corresponding outputs
- The rows of the table (number of combinations) depends on the number of variables
  - For n variables we have 2^n rows

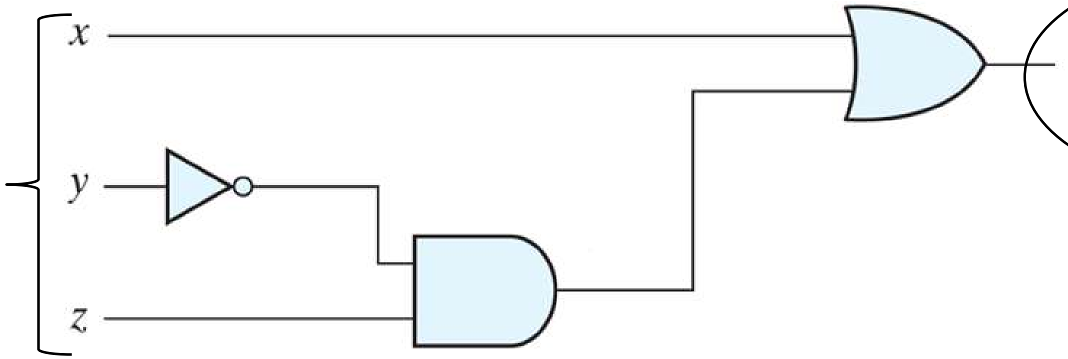
3 input variables

$2^3 = 8\ rows$

x	y	z	F_1	F_2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

Boolean Expression →Truth Table

Input Variables



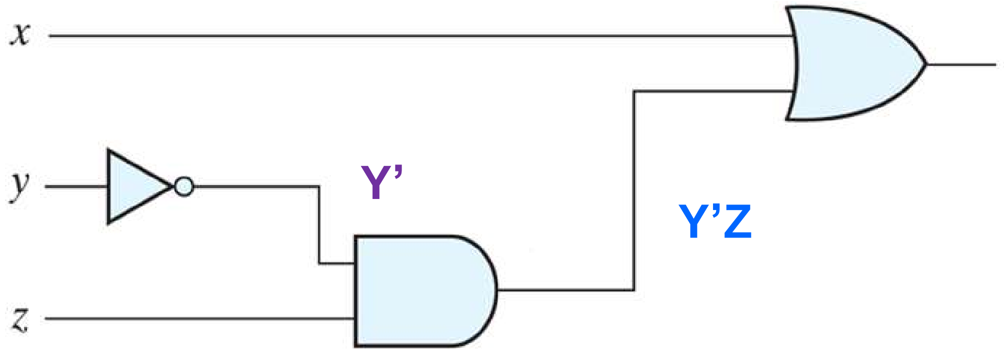
$$F = X + \overline{Y}.Z$$

Boolean Expression

Derive the Truth Table.

X	Y	Z		F
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Boolean Expression → Truth Table

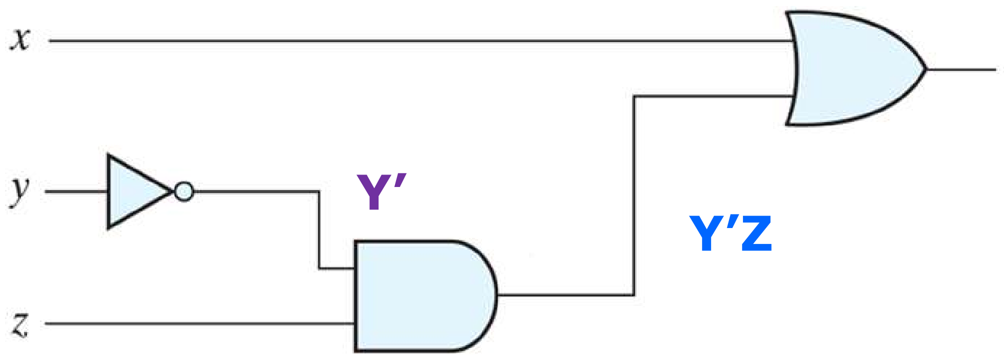


$$F = X + \overline{Y}.Z$$

First Get the expression of the intermediate variables

X	Y	Z	Y'	Y'Z	F
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Boolean Expression → Truth Table

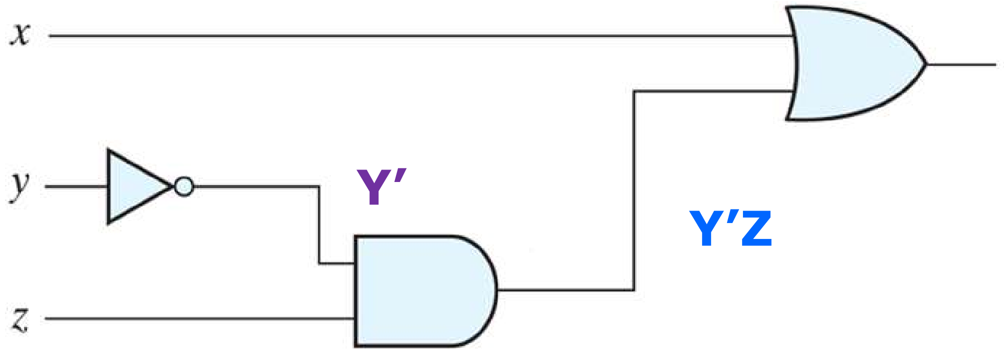


$$F = X + \overline{Y}.Z$$

We get the values of $y^{`}$

X	Y	Z	Y'	Y'Z	F
0	0	0	1		
0	0	1	1		
0	1	0	0		
0	1	1	0		
1	0	0	1		
1	0	1	1		
1	1	0	0		
1	1	1	0		

Boolean Expression → Truth Table

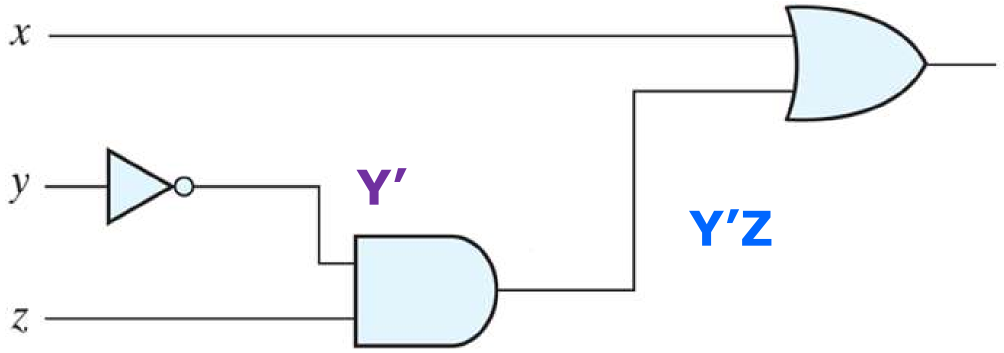


$$F = X + \overline{Y}.Z$$

Then the values of $y`z$

X	Y	Z	Y'	Y'Z	F
0	0	0	1	0	
0	0	1	1	1	
0	1	0	0	0	
0	1	1	0	0	
1	0	0	1	0	
1	0	1	1	1	
1	1	0	0	0	
1	1	1	0	0	

Boolean Expression → Truth Table



$$F = X + \overline{Y}.Z$$

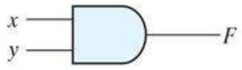
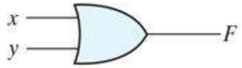
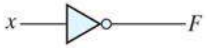
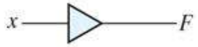
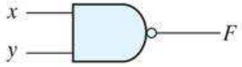
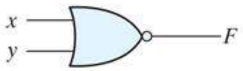
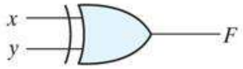
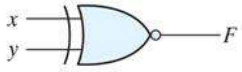
And finally, the value of the function F

X	Y	Z	Y’	Y’Z	F
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

- Logic Gates

Common Logic Gates and Expressi

Boolean Functions	Operator Symbol	Name	Comments
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x, but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y, but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y, but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$(x \oplus y)'$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	x y F 0 0 0 0 1 0 1 0 0 1 1 1
OR		$F = x + y$	x y F 0 0 0 0 1 1 1 0 1 1 1 1
Inverter		$F = x'$	x F 0 1 1 0
Buffer		$F = x$	x F 0 0 1 1
NAND		$F = (xy)'$	x y F 0 0 1 0 1 1 1 0 1 1 1 0
NOR		$F = (x + y)'$	x y F 0 0 1 0 1 0 1 0 0 1 1 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	x y F 0 0 0 0 1 1 1 0 1 1 1 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	x y F 0 0 1 0 1 0 1 0 0 1 1 1

Boolean Operator Precedence

- The order of evaluation in a Boolean expression is:

1. Parentheses 2. Not

3. And 4. Or

- Consequence: Parentheses appear around OR expressions
- Example: $F = A(B + C)(C + \overline{D})$

- Compliment of functions

Complement of a Function

<u>The complement</u> representation for a function F, (F’), is obtained from an ***<u>interchange</u>*** of 1’s to 0’s and 0’s to 1’s ***<u>for the values of F</u>*** in the truth table.

$$F = \overline{X}Y\overline{Z} + \overline{X}\,\overline{Y}Z$$

X	Y	Z	F	$\overline{F}$
0	0	0	**0**	**1**
0	0	1	**1**	**0**
0	1	0	**1**	**0**
0	1	1	**0**	**1**
1	0	0	**0**	**1**
1	0	1	**0**	**1**
1	1	0	**0**	**1**
1	1	1	**0**	**1**

$$\overline{F} = \overline{X}\cdot\overline{Y}\cdot\overline{Z} + \overline{X}\cdot Y\cdot Z + X\cdot\overline{Y}\cdot\overline{Z} + X\cdot\overline{Y}\cdot Z + X\cdot Y\cdot\overline{Z} + X\cdot Y\cdot Z$$

Complement of a Function … Cont.

- **<u>The complement of a function</u>** can be derived algebraically by applying DeMorgan's theorem.
- Complement the <u>left hand side</u> and the <u>right hand side</u> by putting a bar on top of each.

$$F = \overline{X}Y\overline{Z} + \overline{X}\,\overline{Y}Z$$

$$\overline{F} = \overline{\overline{X}Y\overline{Z} + \overline{X}\,\overline{Y}Z}$$

$$\overline{F} = (\overline{\overline{X}Y\overline{Z}}).(\overline{\overline{X}\,\overline{Y}Z})$$

$$\overline{F} = (X + \overline{Y} + Z)(X + Y + \overline{Z})$$

Complement of a Function … Cont.

<u>Yet another way</u> of deriving the complement of a function is to:

I. Take the dual of the function equation

II. Complement each literal.

$$F = \overline{X}Y\overline{Z} + \overline{X}\,\overline{Y}Z$$

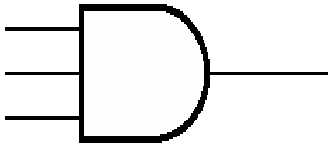
$$F = (\overline{X}Y\overline{Z}) + (\overline{X}\,\overline{Y}Z)$$

$$Dual \Rightarrow (\overline{X} + Y + \overline{Z}).(\overline{X} + \overline{Y} + Z)$$

$$\overline{F} = (X + \overline{Y} + Z).(X + Y + \overline{Z})$$

- Boolean Expression Simplification

Representation: Truth Table



- Give Boolean Expression?

$$F = \overline{X} \bullet \overline{Y} \bullet Z + X \bullet \overline{Y} \bullet \overline{Z} + X \bullet \overline{Y} \bullet Z + X \bullet Y \bullet \overline{Z} + X \bullet Y \bullet Z$$

- Not Simplified!
- Can we do better

2^n rows where n is # of variables

X	Y	Z		F
0	0	0		0
0	0	1	$\overline{X} \bullet \overline{Y} \bullet Z$	1
0	1	0		0
0	1	1		0
1	0	0	$X \bullet \overline{Y} \bullet \overline{Z}$	1
1	0	1	$X \bullet \overline{Y} \bullet Z$	1
1	1	0	$X \bullet Y \bullet \overline{Z}$	1
1	1	1	$X \bullet Y \bullet Z$	1

Truth Table → Boolean Function

- Let us first reconsider how to get a Boolean expression from a truth table
- Consider a truth table
- The Boolean expression for F:
  - Sum or minterms (Oring of terms)
  - Product of maxterms (Anding of terms)
- What is a minterm?
- What is a maxterm?

Truth Table for the Function $F = X + \overline{Y}Z$

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

- Min Terms

From Truth Table to Function

- Consider a truth table
- Can implement F by taking OR of all terms that are 1

$$F = \overline{X}\overline{Y}Z$$

MinTerm

$$F = \overline{X} \cdot \overline{Y} \cdot Z + X \cdot \overline{Y} \cdot \overline{Z} + X \cdot \overline{Y} \cdot Z + X \cdot Y \cdot \overline{Z} + X \cdot Y \cdot Z$$

$$F(optimized) = \overline{Y}Z + X$$

Product Term

Truth Table
for the Function $F = X + \overline{Y}Z$

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Number of Minterms

- For *n* variables, there will be 2^n minterms
- Like binary numbers from 0 to 2^n-1

3-variables → 8 minterms

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

2-variables → 4 minterms

X	Y	F
0	0	1
0	1	0
1	0	0
1	1	0

Product term

- Assume we have a circuit with 3 input variables X, Y, Z
- Definitions:
  - Product Term – a <u>subset</u> of the variables appear (complimented or not with an AND operation between them)

    → XY, XZ, XZ’, XYZ
  - <u>Min Term</u> – is a <u>product term</u> in which <u>*all variables*</u> appear once → (XYZ, X’YZ, X’Y’Z’)
- All min-terms are product terms but not all product terms are min terms

Definition: Minterm

Is a **Product Term** in which **ALL** variables appear once (complemented or not)

One minterm active at a time

X	Y	Z	Product Term	Symbol	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\overline{X}\,\overline{Y}\,\overline{Z}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\overline{X}\,\overline{Y}Z$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\overline{X}Y\overline{Z}$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\overline{X}YZ$	m_3	0	0	0	1	0	0	0	0
1	0	0	$X\overline{Y}\,\overline{Z}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$X\overline{Y}Z$	m_5	0	0	0	0	0	1	0	0
1	1	0	$XY\overline{Z}$	m_6	0	0	0	0	0	0	1	0
1	1	1	XYZ	m_7	0	0	0	0	0	0	0	1

Sum of Minterms (SOM)

OR all of the minterms of truth table row with a 1

$$F = \overline{X}\cdot\overline{Y}\cdot\overline{Z} + \overline{X}\cdot Y\cdot\overline{Z} + X\cdot\overline{Y}\cdot Z + X\cdot Y\cdot Z$$

In this case $m_0+m_2+m_5+m_7$

$$\overline{F} = \overline{X}\,\overline{Y}Z + \overline{X}YZ + X\overline{Y}\,\overline{Z} + XY\overline{Z}$$

In this case $m_1 + m_3 + m_4 + m_6$

	X	Y	Z	F	$\overline{F}$
m_0	0	0	0	1	0
m_1	0	0	1	0	1
m_2	0	1	0	1	0
m_3	0	1	1	0	1
m_4	1	0	0	0	1
m_5	1	0	1	1	0
m_6	1	1	0	0	1
m_7	1	1	1	1	0

Alternative Representation

$$F = \overline{X}.\overline{Y}.\overline{Z} + \overline{X}.Y.\overline{Z} + X.\overline{Y}.Z + X.Y.Z$$

In this case $m_0+m_2+m_5+m_7$

$$F(X,Y,Z) = \sum m(0,2,5,7)$$

$$\overline{F} = \overline{X}.\overline{Y}.Z + \overline{X}.Y.Z + X.\overline{Y}.\overline{Z} + X.Y.\overline{Z}$$

In this case $m_1 + m_3 + m_4 + m_6$

$$\overline{F}(X,Y,Z) = \sum m(1,3,4,6)$$

X	Y	Z	F	$\overline{F}$
0	0	0	1	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

Summary of Properties of Minterms

- There are 2^n minterms for a Boolean function with n variables
- Any Boolean function can be expressed as a logical sum of minterms
- The complement of a function contains those minterms not included in the original function.
- A function that includes all the 2^n minterms is equal to logic 1.

- Sum of Products

Sum of Products (SOP) Standard Form

- The sum-of-minterms form is a **canonical algebraic** expression that is obtained **directly from a truth table**.
- The expression obtained contains the **maximum number of literals** in each term.
- ***Simplification of the sum-of-minterms*** expression is called **sum-of-products.**
  - Simplified expression is called a **standard form**

SOM vs. SOP

Sum of minterms (SOM)

$$F = \overline{X} \cdot \overline{Y} \cdot \overline{Z} + \overline{X} \cdot Y \cdot \overline{Z} + X \cdot \overline{Y} \cdot Z + X \cdot Y \cdot Z$$

X	Y	Z	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

After Simplification we get the Sum of products (SOP)

$$F = \overline{X}.\overline{Z} + X.Z$$

Sum of Products Implementation

Sum of products

$$F = \overline{Y} + \overline{X}Y\overline{Z} + XY$$

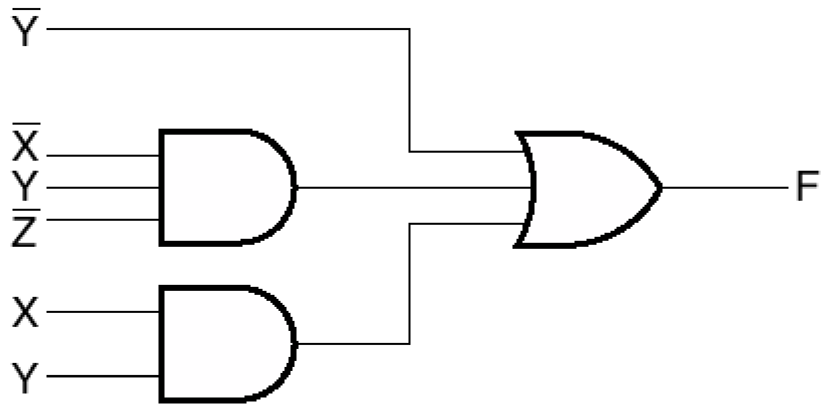
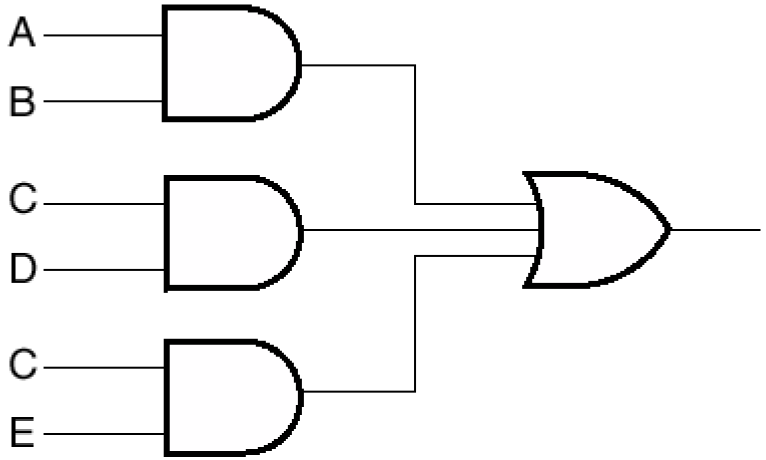


Fig. 2-5 Sum-of-Products Implementation

We refer to this implementation as a two-level circuit

2-level Implementation

- Sum of products has 2 levels of gates



(b) AB + CD + CE

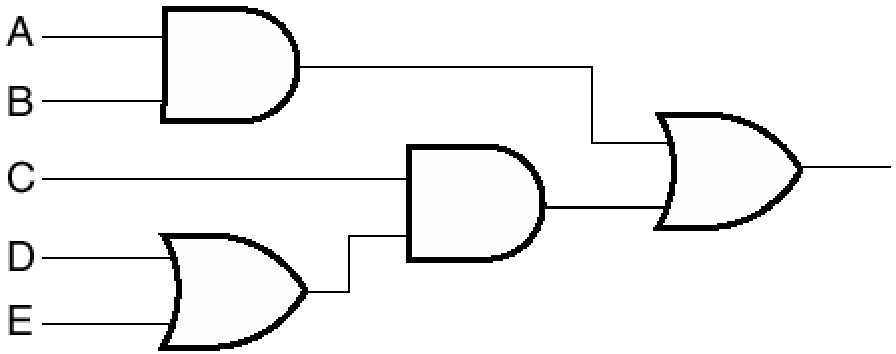
- **Is there a 3-level rep of this circuit?**

2-level vs. 3-level Implementation

AB + CD + CE can be

Also expressed as

AB + C(D+E)



(a) AB + C (D + E)

What's best?

- **Hard to answer!!**
- **More gate delays?**
- **But maybe we only have 2-input gates**

- Max term

Definition: Maxterm

Is a **Sum Term** in which **all** variables appear once (complemented or not)

X	Y	Z	Sum Term	Symbol	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	0	0	$X+Y+Z$	M_0	0	1	1	1	1	1	1	1
0	0	1	$X+Y+\overline{Z}$	M_1	1	0	1	1	1	1	1	1
0	1	0	$X+\overline{Y}+Z$	M_2	1	1	0	1	1	1	1	1
0	1	1	$X+\overline{Y}+\overline{Z}$	M_3	1	1	1	0	1	1	1	1
1	0	0	$\overline{X}+Y+Z$	M_4	1	1	1	1	0	1	1	1
1	0	1	$\overline{X}+Y+\overline{Z}$	M_5	1	1	1	1	1	0	1	1
1	1	0	$\overline{X}+\overline{Y}+Z$	M_6	1	1	1	1	1	1	0	1
1	1	1	$\overline{X}+\overline{Y}+\overline{Z}$	M_7	1	1	1	1	1	1	1	0

Minterm/Maxterm: Relationship

- Minterm and maxterm with same subscripts are complements

$$\overline{m}_j = Mj$$

- Example (Use DeMorgan's theory)

$$m_3 = \overline{X}YZ$$
$$\overline{m}_3 = \overline{\overline{X}YZ} = X + \overline{Y} + \overline{Z} = M_3$$

Product of Maxterms

We can also express F as ANDing of all rows that should evaluate to 0

$$F = M_1 \bullet M_3 \bullet M_4 \bullet M_6$$

$$F(X,Y,Z) = \Pi M(1,3,4,6)$$

$$F = (X + Y + \bar{Z})(X + \bar{Y} + \bar{Z})$$
$$(\bar{X} + Y + Z)(\bar{X} + \bar{Y} + Z)$$

	X	Y	Z	F
M_0	0	0	0	1
M_1	0	0	1	$\Rightarrow$ 0
M_2	0	1	0	1
M_3	0	1	1	$\Rightarrow$ 0
M_4	1	0	0	$\Rightarrow$ 0
M_5	1	0	1	1
M_6	1	1	0	$\Rightarrow$ 0
M_7	1	1	1	1

- Expression Simplification

Useful Boolean Algebra thermos for expression manipulation and simplification

- Table 2.1

Postulate 2	(a)	$x + 0 = x$	(b)	$x \cdot 1 = x$
Postulate 5	(a)	$x + x' = 1$	(b)	$x \cdot x' = 0$
Theorem 1	(a)	$x + x = x$	(b)	$x \cdot x = x$
Theorem 2	(a)	$x + 1 = 1$	(b)	$x \cdot 0 = 0$
Theorem 3, involution		$(x')' = x$		
Postulate 3, commutative	(a)	$x + y = y + x$	(b)	$xy = yx$
Theorem 4, associative	(a)	$x + (y + z) = (x + y) + z$	(b)	$x(yz) = (xy)z$
Postulate 4, distributive	(a)	$x(y + z) = xy + xz$	(b)	$x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a)	$(x + y)' = x'y'$	(b)	$(xy)' = x' + y'$
Theorem 6, absorption	(a)	$x + xy = x$	(b)	$x(x + y) = x$

Useful Boolean Algebra thermos for expression manipulation and simplification

- **Dual of A function**
- The dual of an expression is obtained by:
  1. Changing AND to OR, and
  2. Changing OR to AND throughout
  3. Changing 1's to 0's, and
  4. Changing 0's to 1's

- For example
  - The dual of X+0 is:
    - X.1,
  - The dual of X.0 is:
    - X+1

Example 1: Boolean Algebraic Proof

- $A + A \cdot B = A$ (Absorption Theorem)

Proof Steps	Justification (identity or theorem)
$A + A \cdot B$	
$= A \cdot 1 + A \cdot B$	$X = X \cdot 1$ → identity #2
$= A \cdot (1 + B)$	$X \cdot Y + X \cdot Z = X \cdot (Y + Z)$ → identity #14
$= A \cdot 1$	$1 + X = 1$ → identity #3
$= A$	$X \cdot 1 = X$ → identity #2

Cont. Boolean Algebraic Proof

Our primary reason for doing proofs is to learn:

- Careful and efficient use of the identities and theorems of Boolean algebra, and

- How to choose the appropriate identity or theorem to apply to make forward progress, irrespective of the application.

- Simplification examples

Algebraic Manipulation

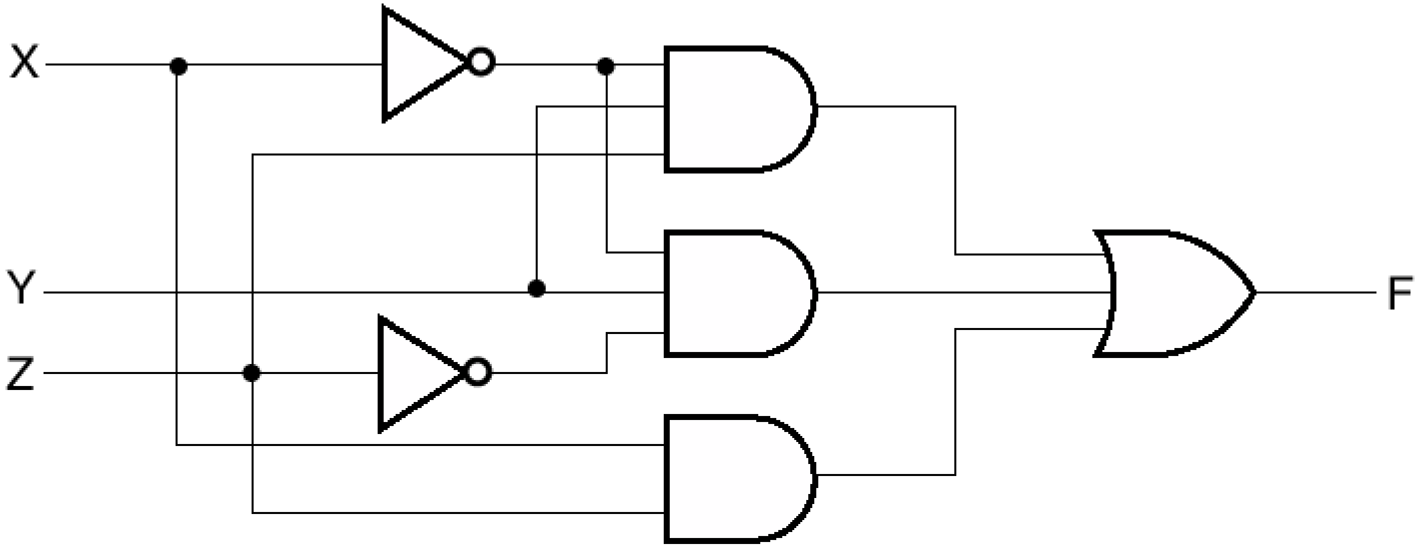
$$F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$$

- When a Boolean equation is implemented with logic gates,
  - i. **Each term requires a gate,**
  - ii. **Each variable designates an input to the gate.**
- We define a literal as a single variable within a term that may or may not be complemented.
- The expression above has 3 terms and 8 literals.

Algebraic Manipulation

- Consider function

$$F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$$



(a) $F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$

How many gates do we need to implement this function?

Simplify Function

$$F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$$

Apply Postulate 4(a) $X(Y+Z) = XY + XZ$

$$F = \overline{X}Y(Z + \overline{Z}) + XZ$$

Apply Postulate 5(a) $X + \overline{X} = 1$

$$F = \overline{X}Y \bullet 1 + XZ$$

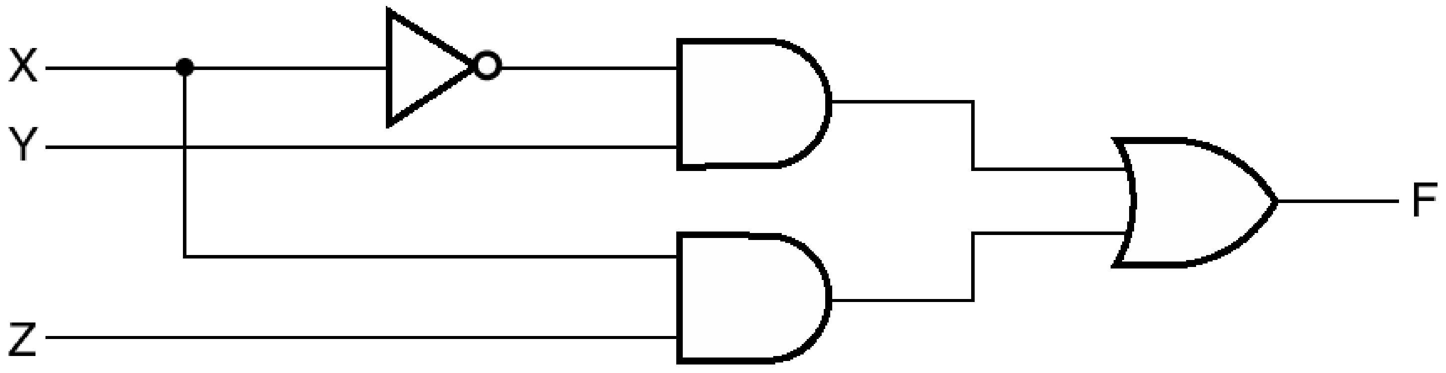
Apply Postulate 2(b) $X \cdot 1 = X$

$$F = \overline{X}Y + XZ$$

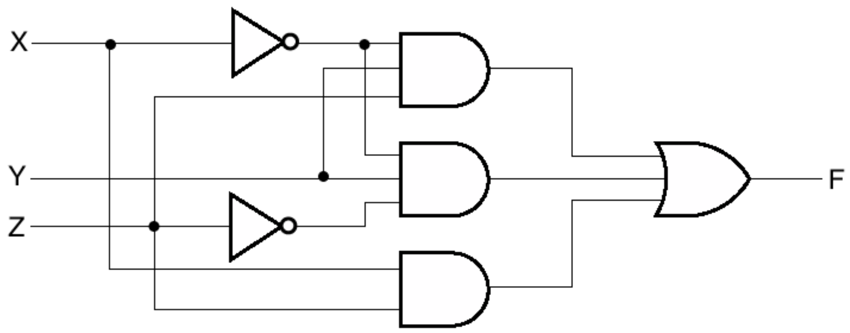
End Result?

Fewer Gates

$$F = \overline{X}Y + XZ$$



(b) $F = \overline{X}Y + XZ$



(a) $F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$

1. **Fewer Gates!**
2. **Fewer Inputs per gate!**

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 5

Gate-Level Minimization

CONTENTS

- The Map Method
- Four-Variable K-Map
- Product-of-Sums Simplification

Weekly Learning Outcomes

1. Know how to derive and simplify a Karnaugh map for Boolean functions of 2, 3, and 4 variables
2. Know how to derive the prime implicants of a Boolean function.
3. Know how to obtain the sum of products and the product of sums forms of a Boolean function directly from its Karnaugh map.
4. Know how to create the Karnaugh map of a Boolean function from its truth table.

Required Reading

1. Chapter 3 (3.1 to 3.4)
(Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapter 3
2. Chapter 4
3. (Digital Logic for Computing) (John Seiffertt Digital Logic for Computing)

Introduction

Gate Level Minimization

- *Gate-level minimization* is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit.
- This task is well understood, but is difficult to execute by manual methods when the logic has more than a few inputs.

The Map Method

The Map Method

- The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented.
- The map method provides a simple, straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method is also known as the *Karnaugh map* or *K-map*.

K-Map

A K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map from the area enclosed by those squares whose minterms are included in the function.

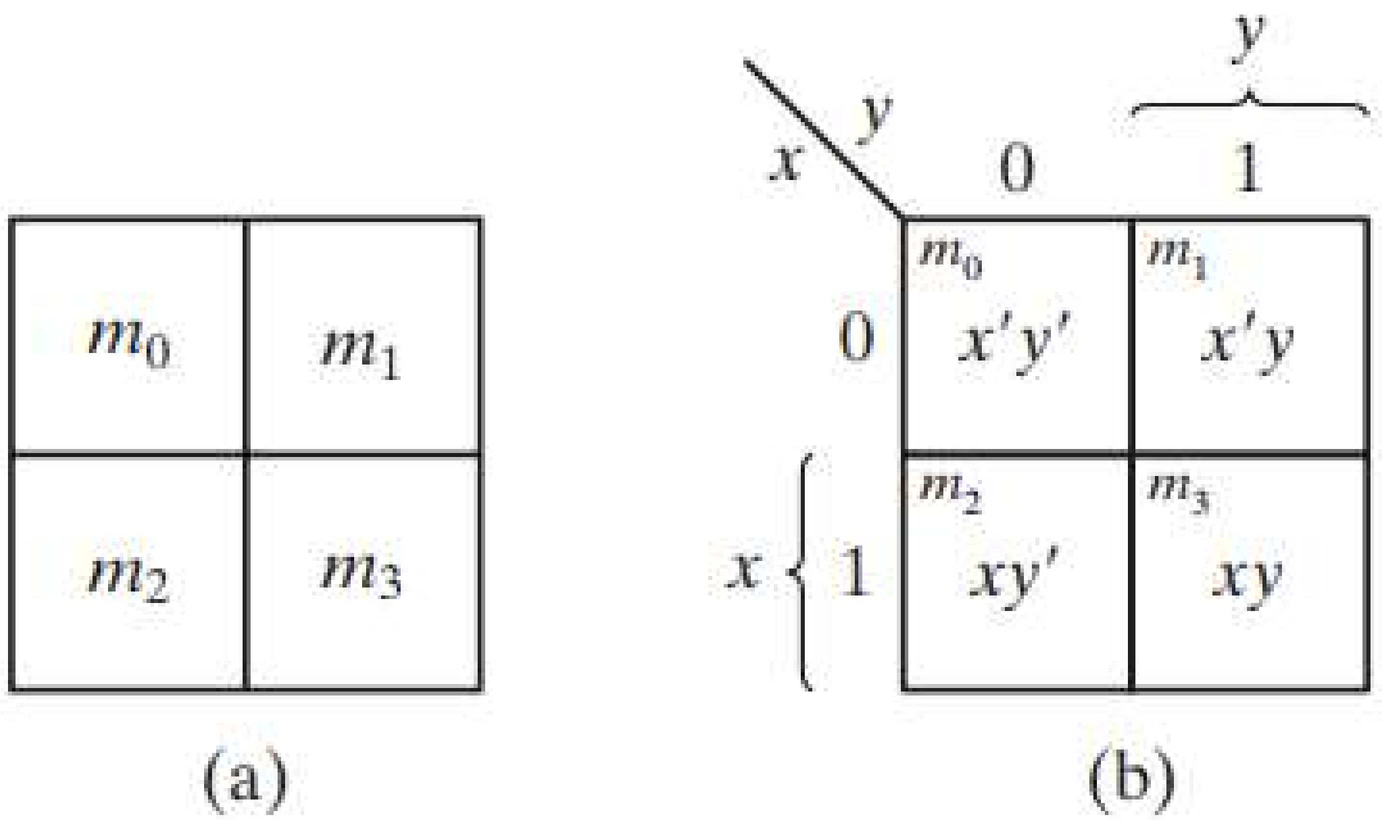
K-Map

- The simplified expressions produced by the map are always in one of the two standard forms: sum of products or product of sums. It will be assumed that the simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals in each term.
- This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate.

Two Variable K-Map

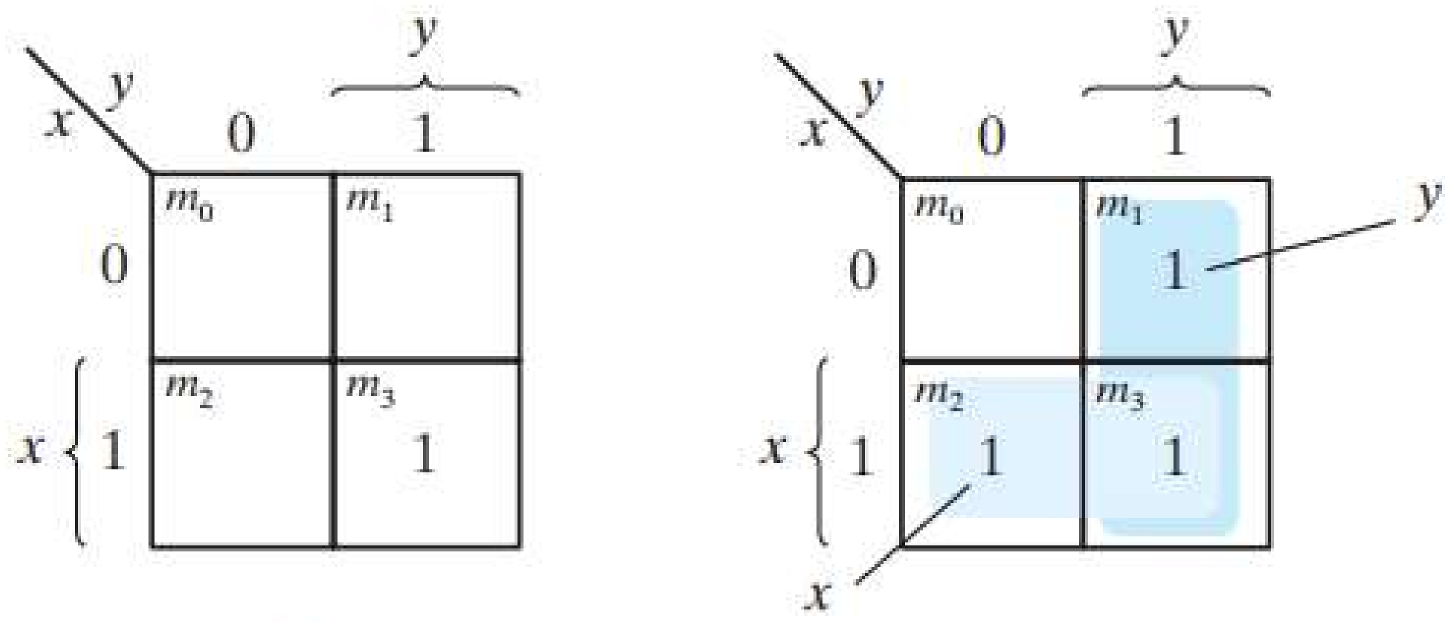
- If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables.
- As an example, the function xy is shown in Fig. 3.2 (a). Since xy is equal to $m3$, a 1 is placed inside the square that belongs to $m3$. Similarly, the function $x + y$ is represented in the map of Fig. 3.2 (b) by three squares marked with 1's. These squares are found from the minterms of the function:

Two Variable K-Map



Two Variable K-Map

$$m_1 + m_2 + m_3 = x'y + xy' + xy = x + y$$



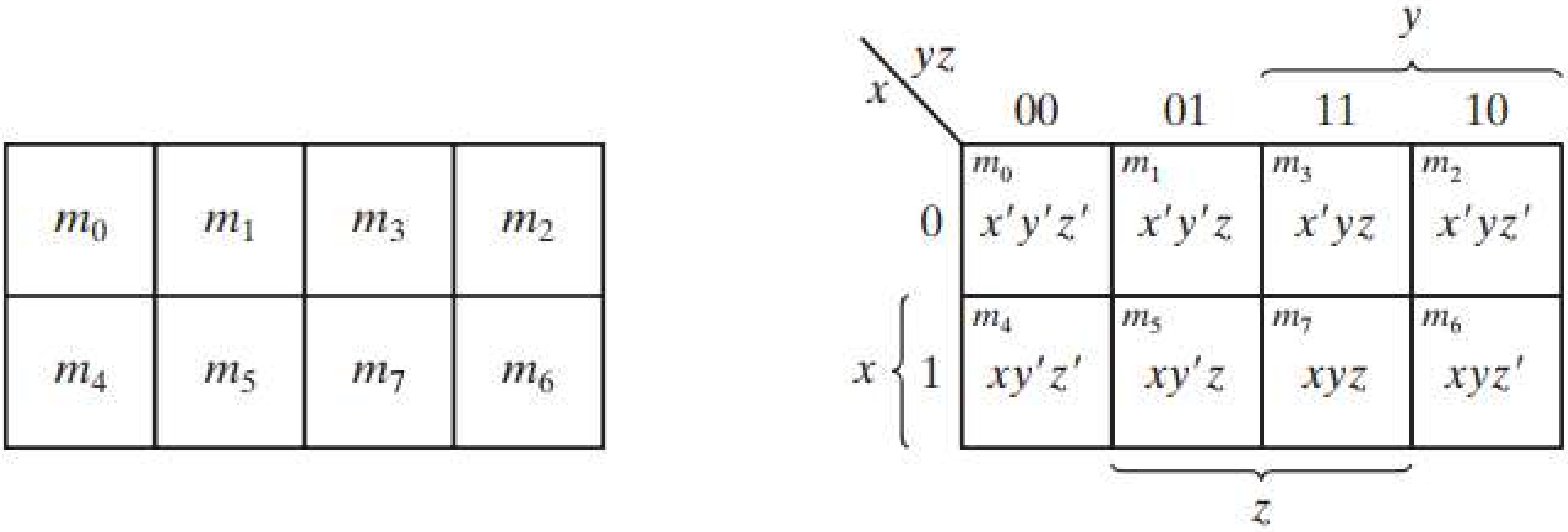
(a) xy (b) $x + y$

Three Variable K-Map

A three-variable K-map is shown below. There are eight minterms for three binary variables; therefore, the map consists of eight squares.

Note that the minterms are arranged, not in a binary sequence, but in a sequence similar to the Gray code. The characteristic of this sequence is that **only one bit changes in value from one adjacent column to the next.

Three Variable K-Map



(a) (b)

We must recognize the basic property possessed by adjacent squares: **Any two adjacent squares in the map differ by only one variable,** which is primed in one square and unprimed in the other.

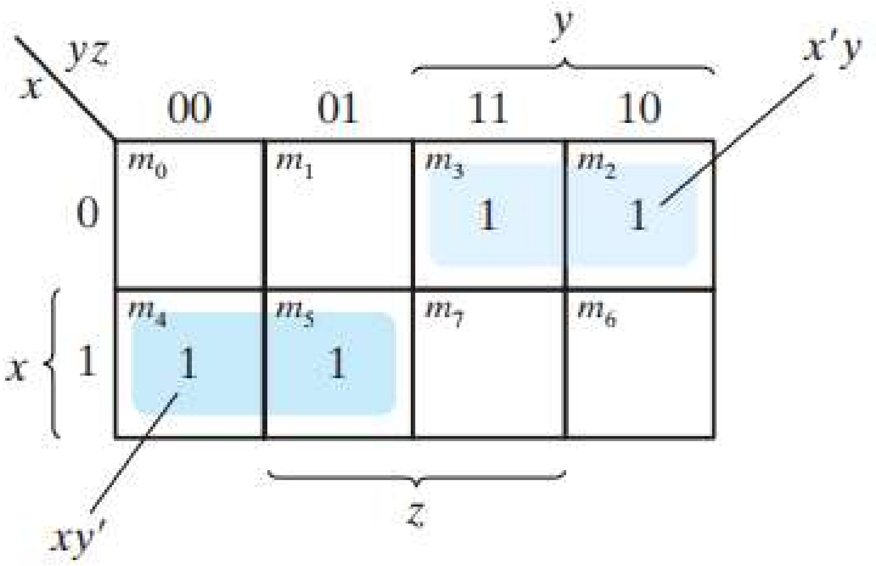
Three Variable K-Map - Example

Simplify the Boolean function

$$F(x, y, z) = \Sigma(2, 3, 4, 5)$$

- First, a 1 is marked in each minterm square that represents the function. This is shown in next slide, in which the squares for minterms 010, 011, 100, and 101 are marked with 1's.
- The next step is to find possible adjacent squares. These are indicated in the map by two shaded rectangles, each enclosing two 1's.
- The upper right rectangle represents the area enclosed by $x'y$. This area is determined by observing that the two-square area is in row 0, corresponding to x', and the last two columns, corresponding to y. Similarly, the lower left rectangle represents the product term xy'.

Three Variable K-Map - Example



$$F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$$

The logical sum of these two product terms gives the simplified expression.

$$F = x'y + xy'$$

Three Variable K-Map - Example

In certain cases, two squares in the map are considered to be adjacent even though they do not touch each other.

- For Example, *m*0 is adjacent to *m*2 and *m*4 is adjacent to *m*6 because their minterms differ by one variable. This difference can be readily verified algebraically:

$$m_0 + m_2 = x'y'z' + x'yz' = x'z'(y' + y) = x'z'$$

$$m_4 + m_6 = xy'z' + xyz' = xz' + (y' + y) = xz'$$

Three Variable K-Map

Consequently, we must modify the definition of adjacent squares to include this and other similar cases. We do so by considering the map as being drawn on a surface in which the right and left edges touch each other to form adjacent squares.

Four Variable K-Map Method

Four Variable K-Map

- The map for Boolean functions of four binary variables (*w*, *x*, *y*, *z*) is shown in next slide.
- In Fig. (a), listed the 16 minterms and the squares assigned to each.
- In Fig. (b), the map is redrawn to show the relationship between the squares and the four variables.
- The rows and columns are numbered in a Gray code sequence, with only one digit changing value between two adjacent rows or columns.

Four Variable K-Map

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

(a)

y

	$wx \backslash yz$	00	01	11	10	
	00	m_0 $w'x'y'z'$	m_1 $w'x'y'z$	m_3 $w'x'yz$	m_2 $w'x'yz'$	
	01	m_4 $w'xy'z'$	m_5 $w'xy'z$	m_7 $w'xyz$	m_6 $w'xyz'$	x
w	11	m_{12} $wxy'z'$	m_{13} $wxy'z$	m_{15} $wxyz$	m_{14} $wxyz'$	
	10	m_8 $wx'y'z'$	m_9 $wx'y'z$	m_{11} $wx'yz$	m_{10} $wx'yz'$	

z

(b)

Four Variable K-Map

The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:

One square represents one minterm, giving a term with four literals.

Two adjacent squares represent a term with three literals.

Four adjacent squares represent a term with two literals.

Eight adjacent squares represent a term with one literal.

Sixteen adjacent squares produce a function that is always equal to 1.

Four Variable K-Map - Example

Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

- The minterms listed in the sum are marked by 1's in the map.

  Eight adjacent squares marked with 1's can be combined to form the one literal term y'.

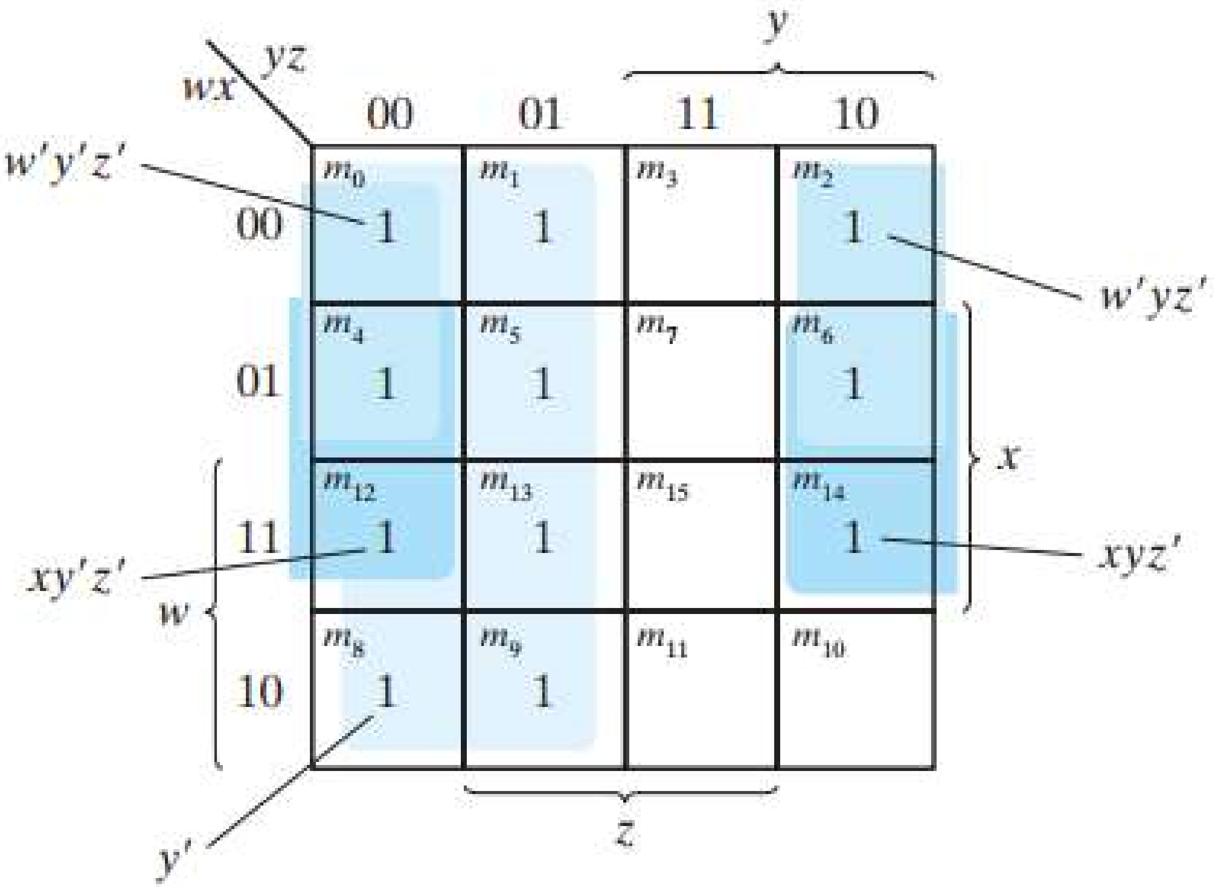
- The top two 1's on the right are combined with the top two 1's on the left to give the term $w'z'$

Four Variable K-Map - Example

- We are now left with a square marked by 1 in the third row and fourth column (square 1110). Instead of taking this square alone (which will give a term with four literals), we combine it with squares already used to form an area of four adjacent squares.
- These squares make up the two middle rows and the two end columns, giving the term xz'
- The simplified function is:

$$F = y' + w'z' + xz'$$

Four Variable K-Map - Example



Note: $w'y'z' + w'yz' = w'z'$

$xy'z' + xyz' = xz'$

Prime Implicants

A *prime implicant* is a product term obtained by combining the maximum possible number of adjacent squares in the map.

In choosing adjacent squares in a map, we must ensure that

(1) all the minterms of the function are covered when we combine the squares

(2) the number of terms in the expression is minimized

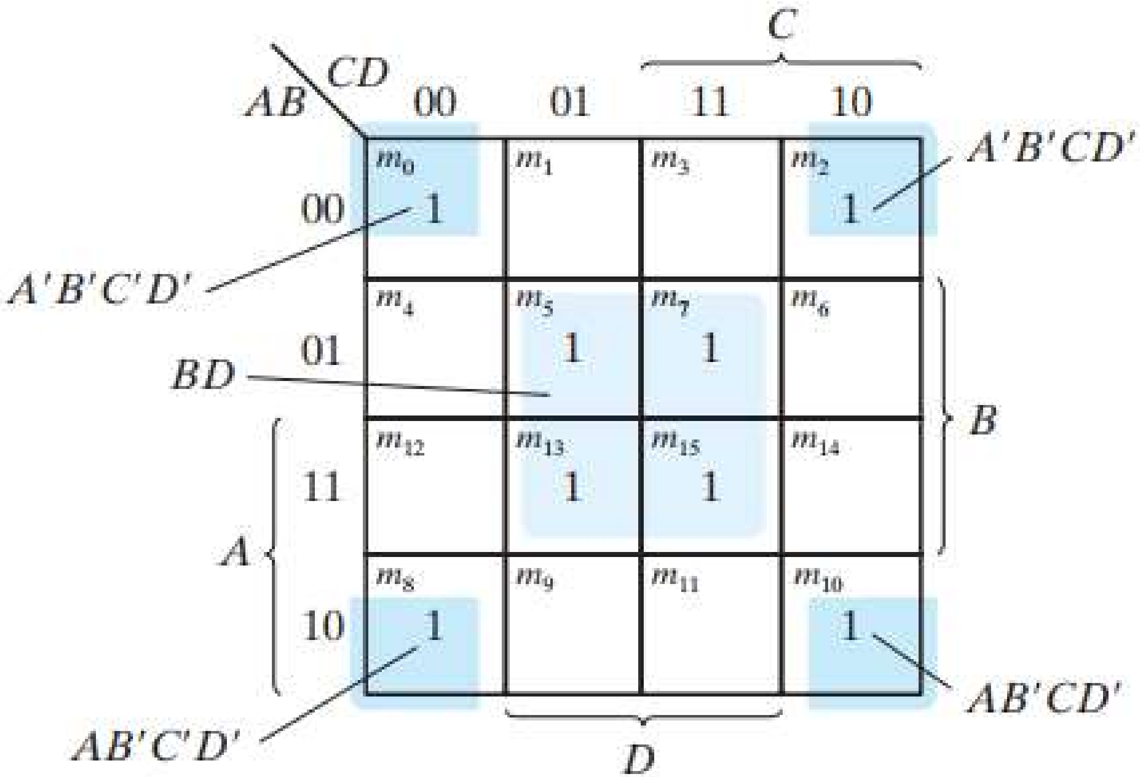
(3) there are no redundant terms (i.e., minterms already covered by other terms)

Prime Implicants

If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be *essential.*

The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares.

Essential Prime Implicants



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$

(a) Essential prime implicants BD and $B'D'$

Higher Variable K-Map

- Maps for more than four variables are not as simple to use as maps for four or fewer variables. A five-variable map needs 32 squares and a six-variable map needs 64 squares.
- When the number of variables becomes large, the number of squares becomes excessive and the geometry for combining adjacent squares becomes more involved.

Product-of-Sums Simplification

Product of Sum Simplification

The minimized Boolean functions derived from the map in all previous examples were expressed in sum-of-products form. With a minor modification, the product-of-sums form can be obtained.

Product of Sum Simplification - Procedure

- The 1’s placed in the squares of the map represent the minterms of the function.
- The minterms not included in the standard sum-of-products form of a function denote the complement of the function.
- The complement of a function is represented in the map by the squares not marked by 1’s.

Product of Sum Simplification - Procedure

- If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified sum-of-products expression of the complement of the function (i.e., of F').
- The complement of F gives us back the function F in product-of-sums form (a consequence of DeMorgan's theorem).

Product of Sum Simplification - Example

Simplify the following Boolean function into

(a) sum-of-products form and

(b) product-of-sums form:

$$F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$$

Product of Sum Simplification - Example

- The 1's marked in the map of Fig (Next Slide) represent all the minterms of the function.

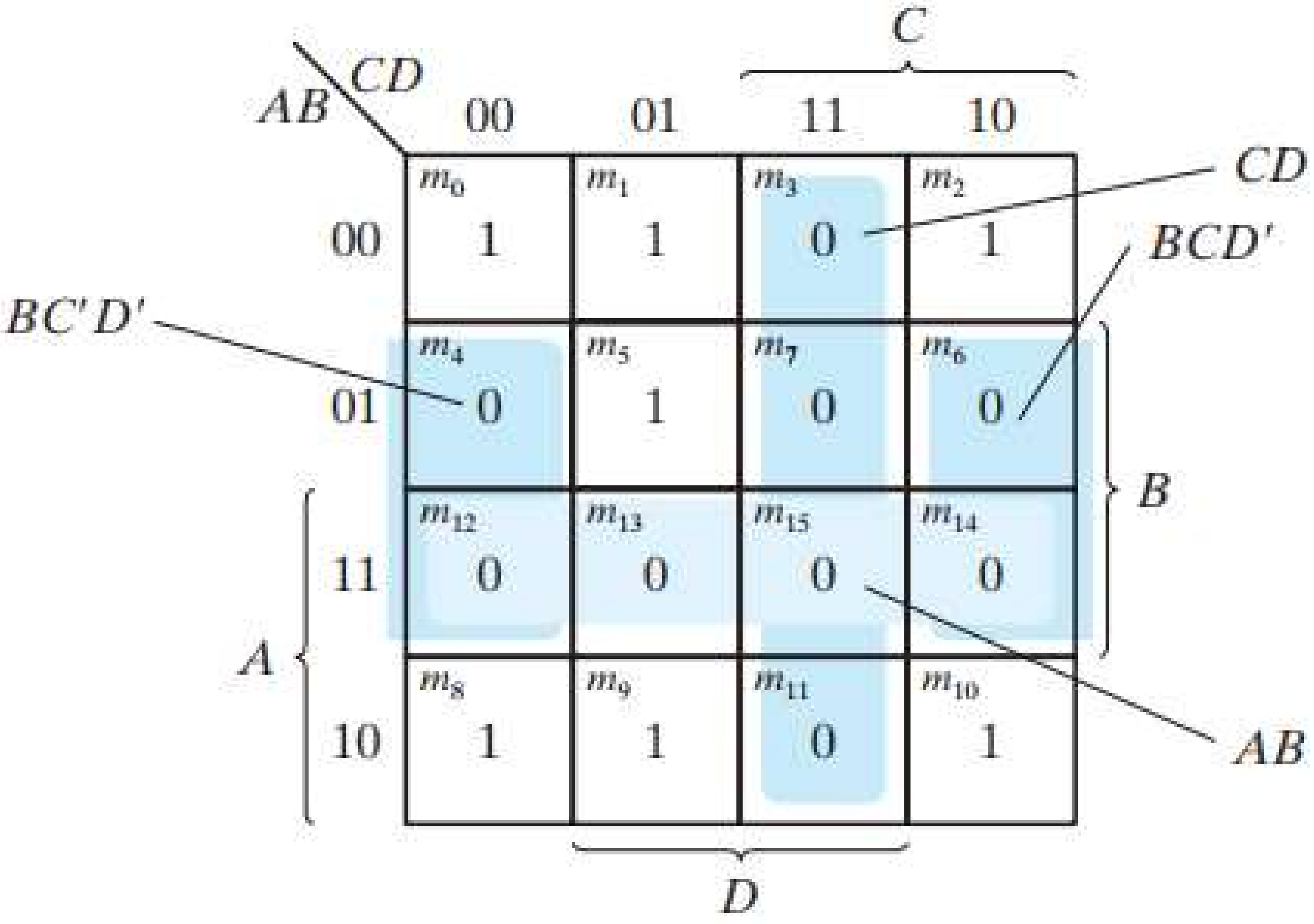
- The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F. Combining the squares with 1's gives the simplified function in sum-of-products form:

  (a) $F = B'D' + B'C' + A'C'D$

  If the squares marked with 0's are combined, as shown in the diagram, we obtain the simplified complemented function:

  $F' = AB + CD + BD'$

Product of Sum Simplification - Example



Note: $BC'D' + BCD' = BD'$

Product of Sum Simplification - Example

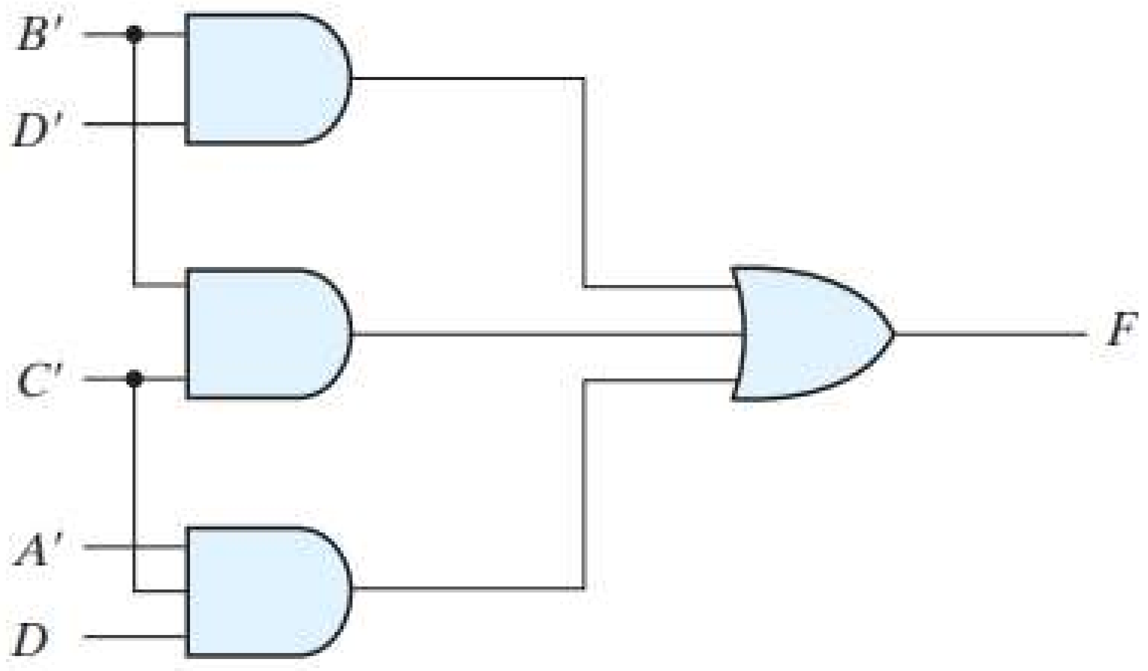
Applying DeMorgan's theorem (by taking the dual and complementing each literal), we obtain the simplified function in product-of-sums form:

(b) $F = (A' + B')(C' + D')(B' + D)$

Product of Sum Simplification - Example

The gate-level implementation of the simplified expressions

(a) The sum-of-products expression

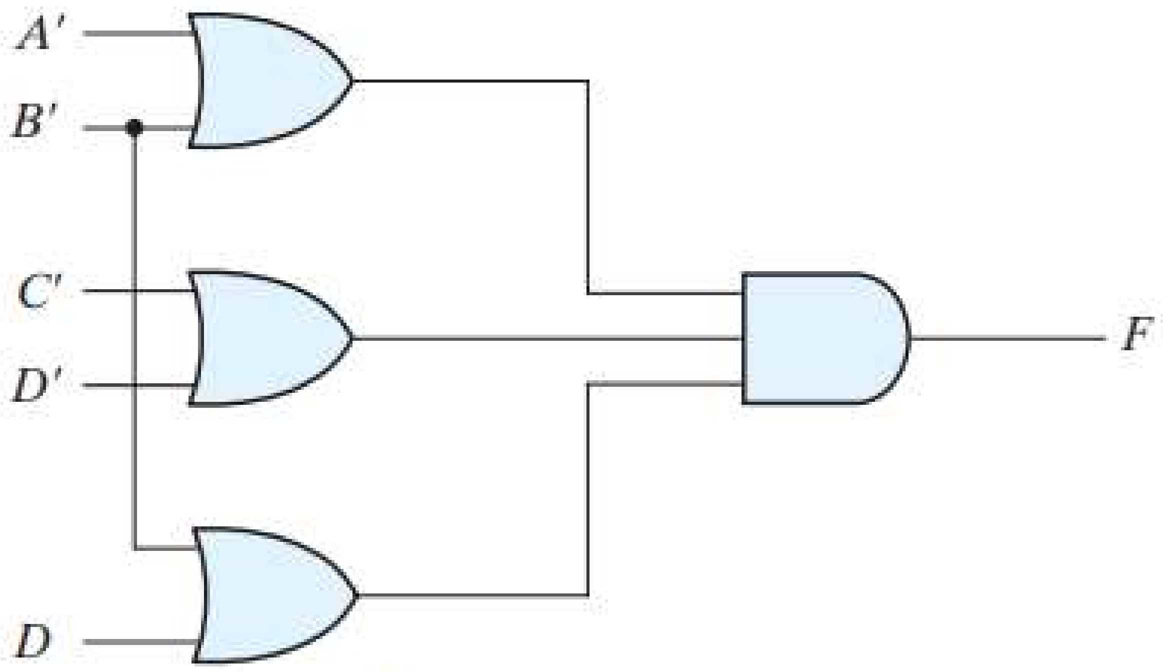


(a) $F = B'D' + B'C' + A'C'D$

Product of Sum Simplification - Example

The gate-level implementation of the simplified expressions

(b) The product-of-sums expression



(b) $F = (A' + B')(C' + D')(B' + D)$

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 6

Gate-Level Minimization

Weekly Learning Outcomes

1. Learn how to simplify a Boolean function using Don’t-Care conditions
2. Learn how implement digital circuits (Boolean functions) using various combinations of logic gates

Required Reading

1. Chapter 3 (Sections 3.5- 3.8)) (Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapters 3 (pp 28-30) (John Seiffertt Digital Logic for Computing)
2. https://www.allaboutcircuits.com/textbook/digital/chpt-3/gate-universality/

Don’t Care

- So far we have dealt with functions that were always either 0 or 1
- Sometimes we have some conditions where we ***don’t care*** what result is
- Example: dealing with BCD
  - With 4-bits we can encode 16 items
  - We only care about first 10 (0000 – 1001)
  - The remaining (1010 – 1111) we don’t care

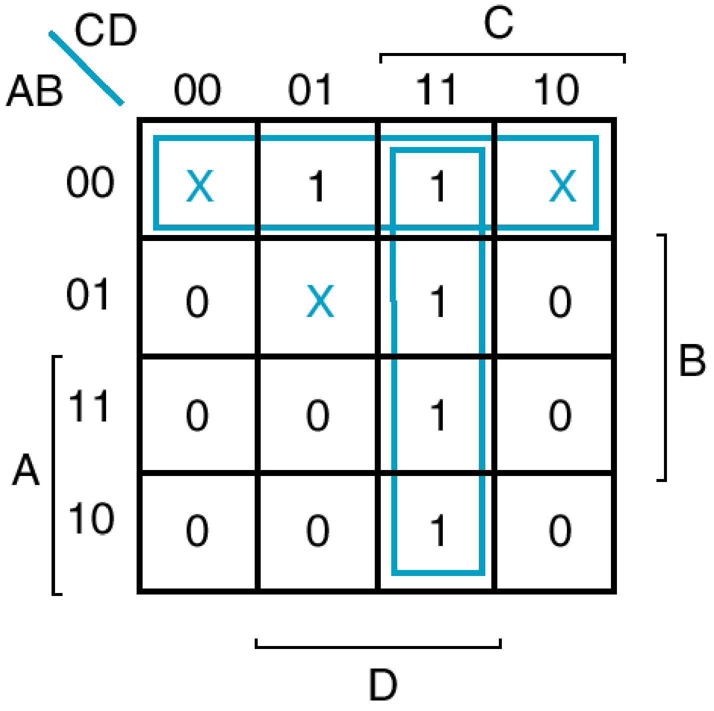
Mark With an `X'

- In a K-map, <u>mark *don't care*</u> with an 'X'
- Leads to <u>simpler implementations</u>
  - Can treat an X either as 1 or 0
- Example:

$$F(A,B,C,D) = \sum m(1,3,7,11,15), d(0,2,5)$$

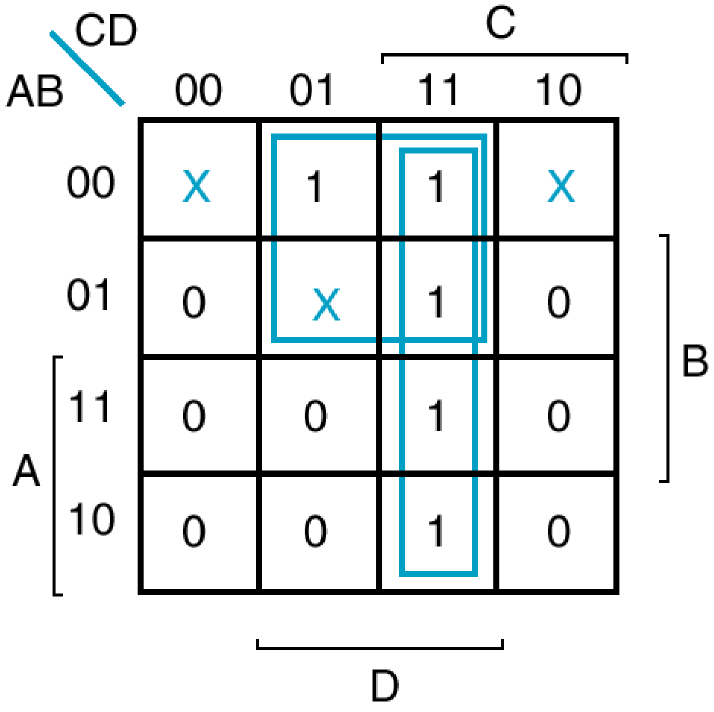
Example

$$F(A,B,C,D)=\sum m(1,3,7,11,15),d(0,2,5)$$



(a) $F = CD + \overline{A}\,\overline{B}$

or



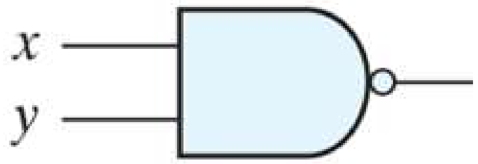
(b) $F = CD + \overline{A}D$

What would we have if Xs were 0?

- NAND and NOR Implementation

NAND Gates

Very common for discrete logic

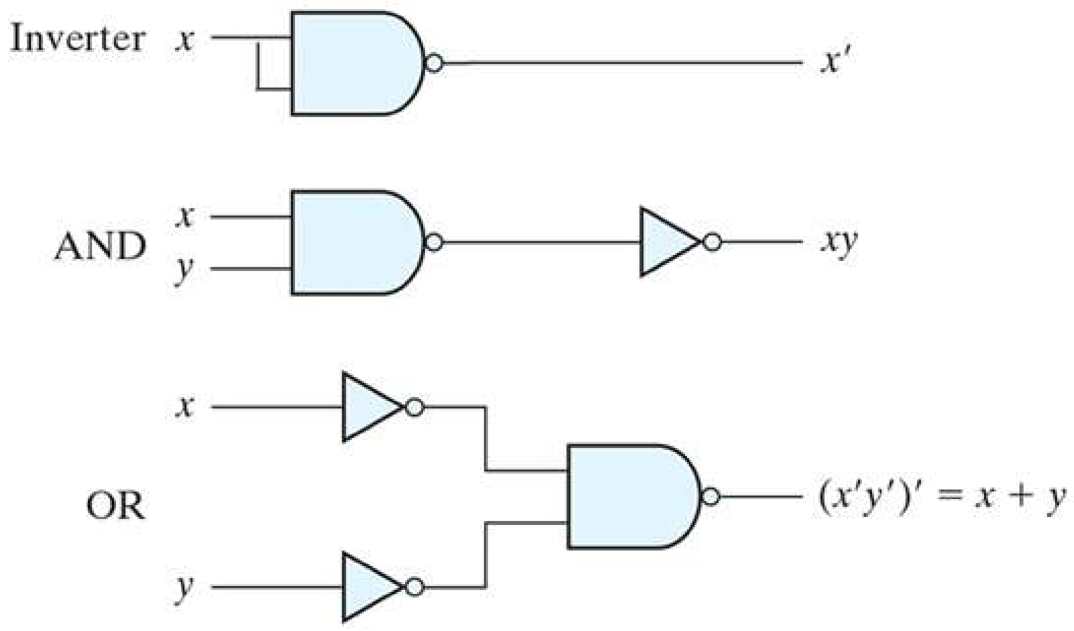


$$F = \overline{X \bullet Y}$$

X	Y	F
0	0	1
0	1	1
1	0	1
1	1	0

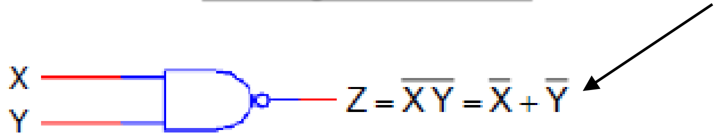
NAND is Universal

- <u>Fact</u>: Any digital circuit <u>can be designed</u> and realized using only AND, OR, NOT gates
- If we can <u>prove</u> that NAND gate <u>can emulate AND, OR, NOT</u>, then we prove that <u>it is Universal</u>



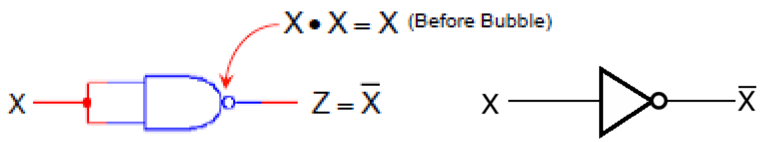
NAND is Universal

<u>DeMorgan's Theorem</u>: Inverting the inputs of an OR gate produces a NAND gate



X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

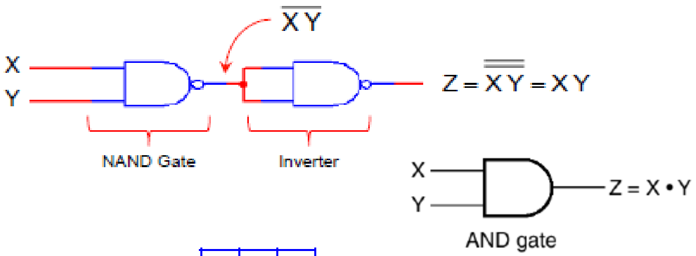
Tie two inputs



X	Z
0	1
1	0

Equivalent to Inverter

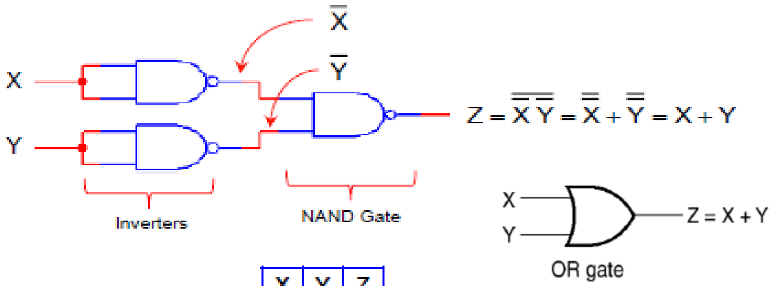
NAND Gate as an AND Gate



X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Equivalent to AND Gate

NAND Gate as an OR Gate



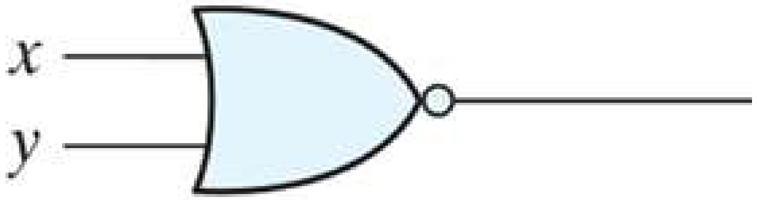
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Equivalent to OR Gate

NOR Gates

- NOT OR
- Also common

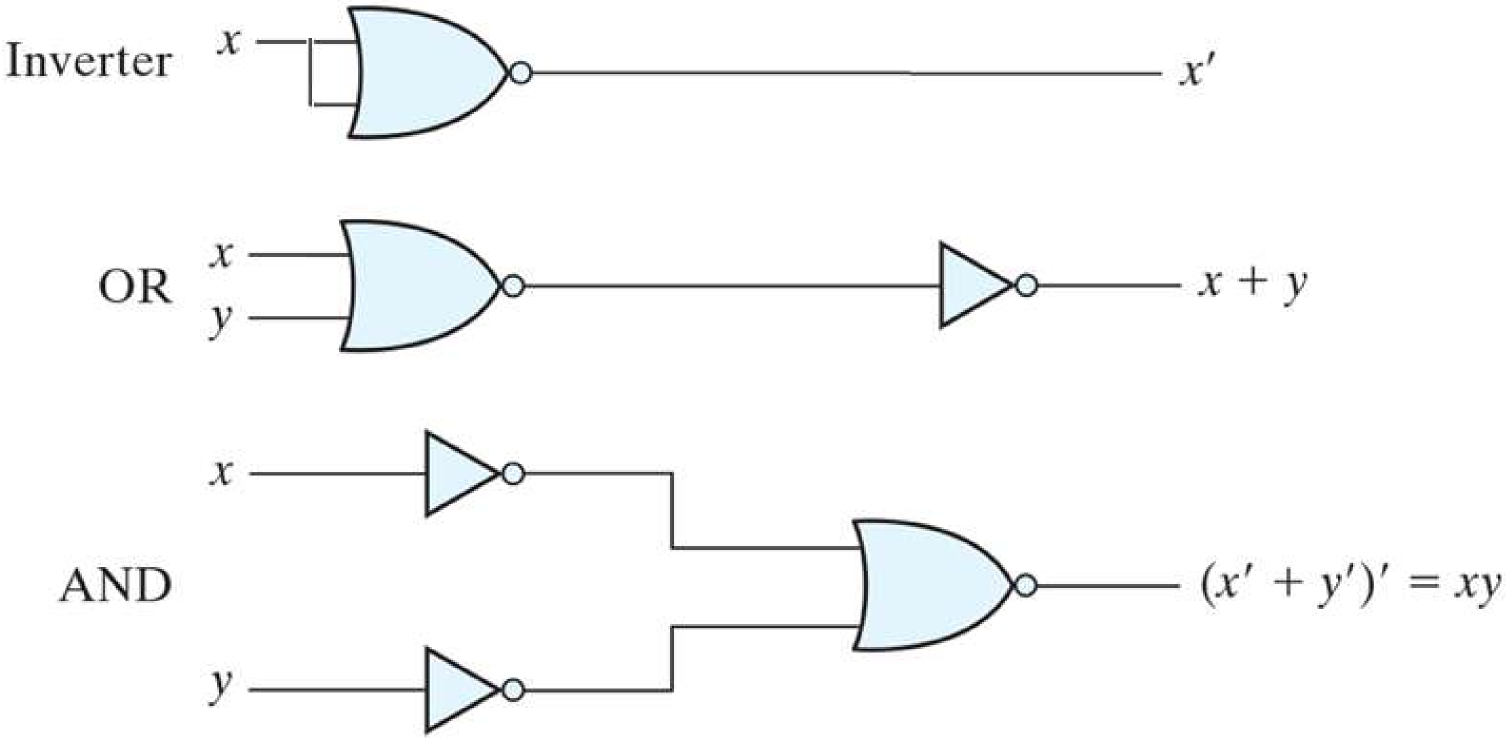
$$F = \overline{X + Y}$$



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

NOR Gate is also Universal

Prove that a NOR gate is Universal by showing it can perform the functionality of (a) NOT gate, (b) OR gate, (c) AND gate



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

- Converting and expression to all-NAND gate

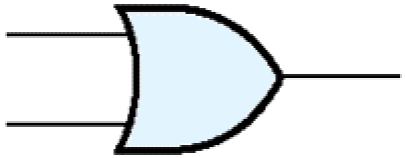
NAND and NOR Implementations

- Digital circuits are frequently constructed with only **NAND** and **NOR** implementations:
  - Both are universal gates
  - they are **easier to fabricate using (CMOS Technology)**

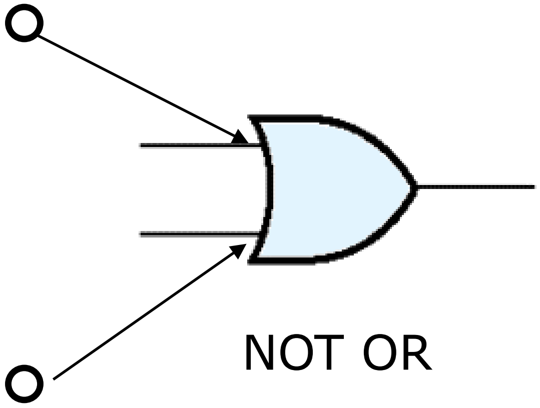
- Because of their use, rules have been developed that allow us to convert Boolean functions using AND, OR and NOT into the equivalent NAND and NOR logic diagrams.

Recall

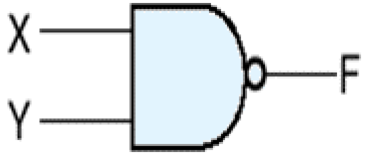
Inverting the inputs of an OR gate produces a NAND gate



OR gate



NOT OR



NAND Gate

OR

X	Y	Z = X + Y
0	0	0
0	1	1
1	0	1
1	1	1

DeMorgan Rule (b)

$$\overline{X.Y} = \overline{X} + \overline{Y}$$

X	Y	F
0	0	1
0	1	1
1	0	1
1	1	0

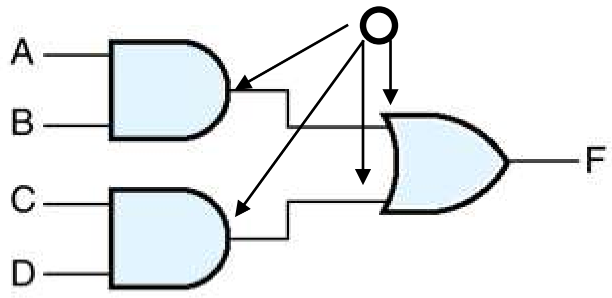
Conversion to all-NAND Circuits

The general procedure for converting a multi-level AND-OR diagram into an all-NAND diagram is as follows:

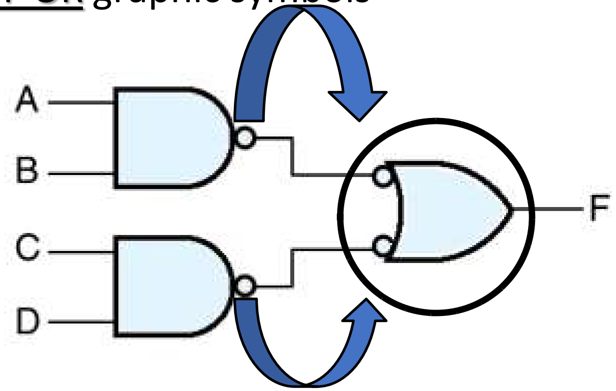
1. Convert all AND gates to NAND gates with AND-NOT graphic symbols
2. Convert all OR gates to NAND gates with NOT-OR graphic symbols
3. Check all the bubbles in the diagram
   - Every bubble that is not compensated by another along the same line will require the insertion of an inverter or complement the input literal

Sum of Products with NAND

- Convert all AND gates to NAND gates with AND-NOT graphic symbols
- Convert all OR gates to NAND gates with NOT-OR graphic symbols

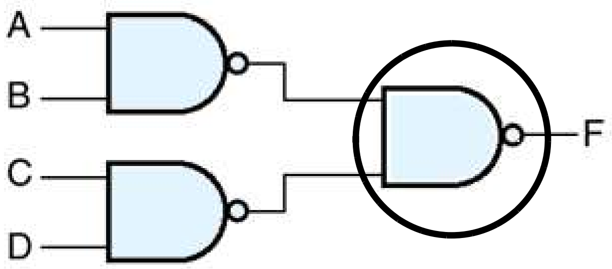


(a)



(b)

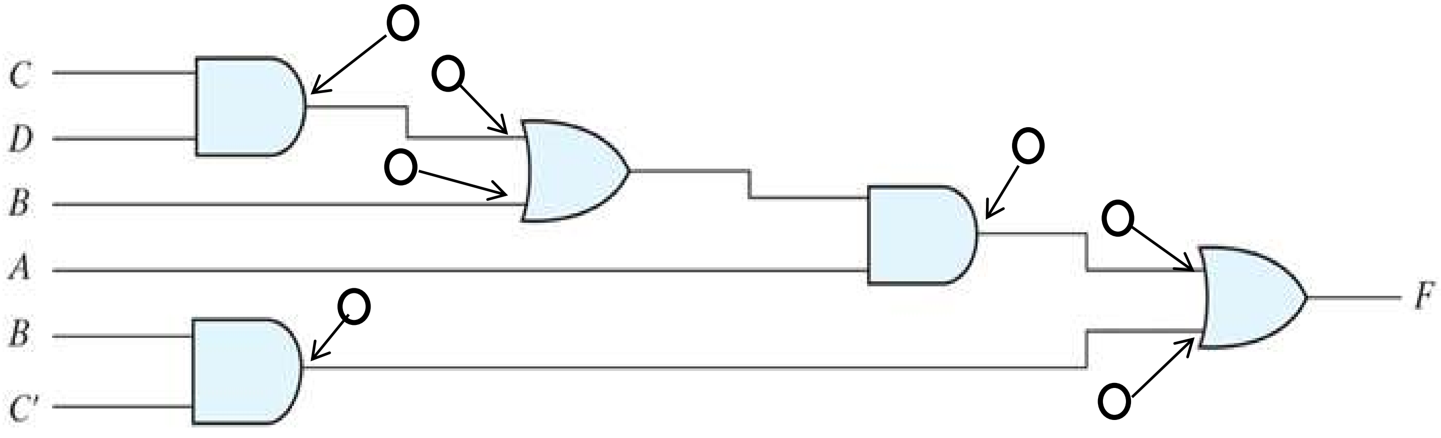
- Check all the bubbles in the diagram
  - Every bubble that is not compensated by another along the same line will require the insertion of an inverter or complement the input literal



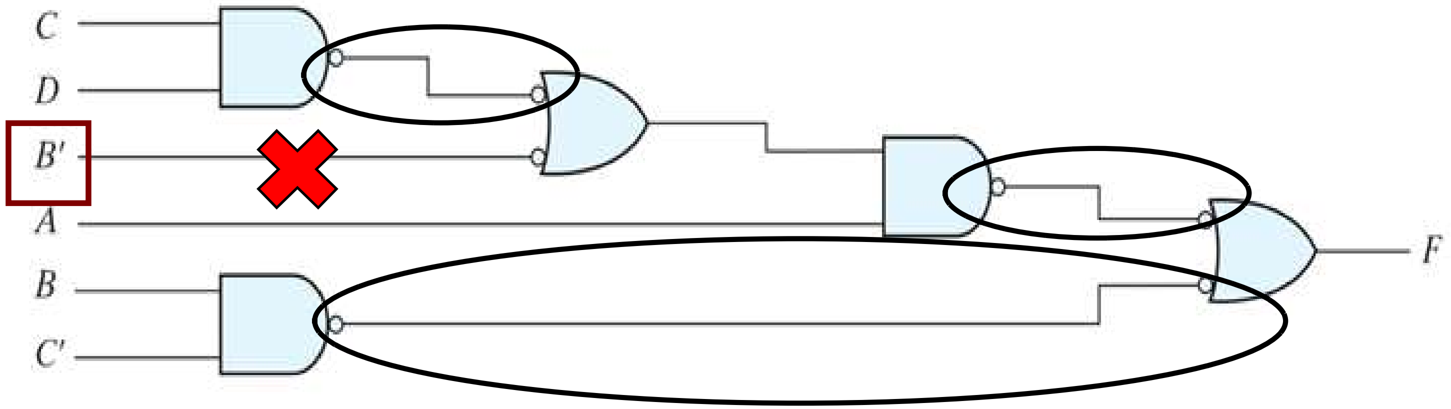
Easy to think of bubbles as canceling

AND-OR Circuit Easy to Convert

Implementing F = A(CD + B) + BC`



(a) AND–OR gates



(b) NAND gates

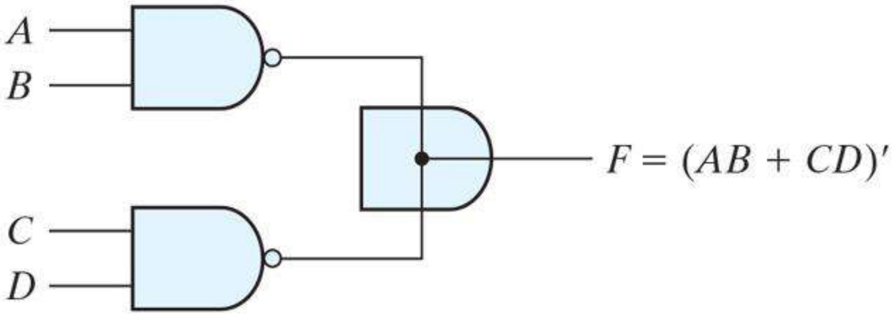
- Other Two-Level Implementations

Wired Logic

- NAND and NOR gates are the most commonly gates found in integrated circuits
- Some implementation allows ***output*** of two NAND or two NOR gates to be ***connected together***
  - This is called ***wired logic***
- Examples
  - AND-OR INVERTER
  - OR-AND-INVERTER
  - ….

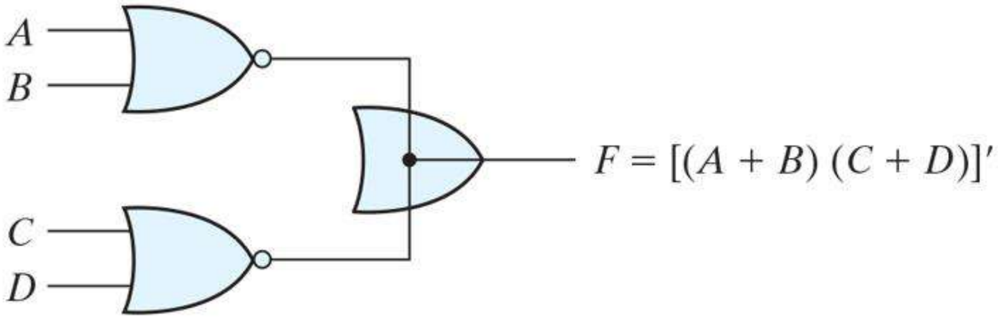
Wired Logic Examples

- Figure 3.26 (Wired logic)



(a) Wired-AND in open-collector
TTL NAND gates.

(AND–OR–INVERT)



(b) Wired-OR in ECL gates

(OR–AND–INVERT)

- We can implement some functions with fewer gates this way

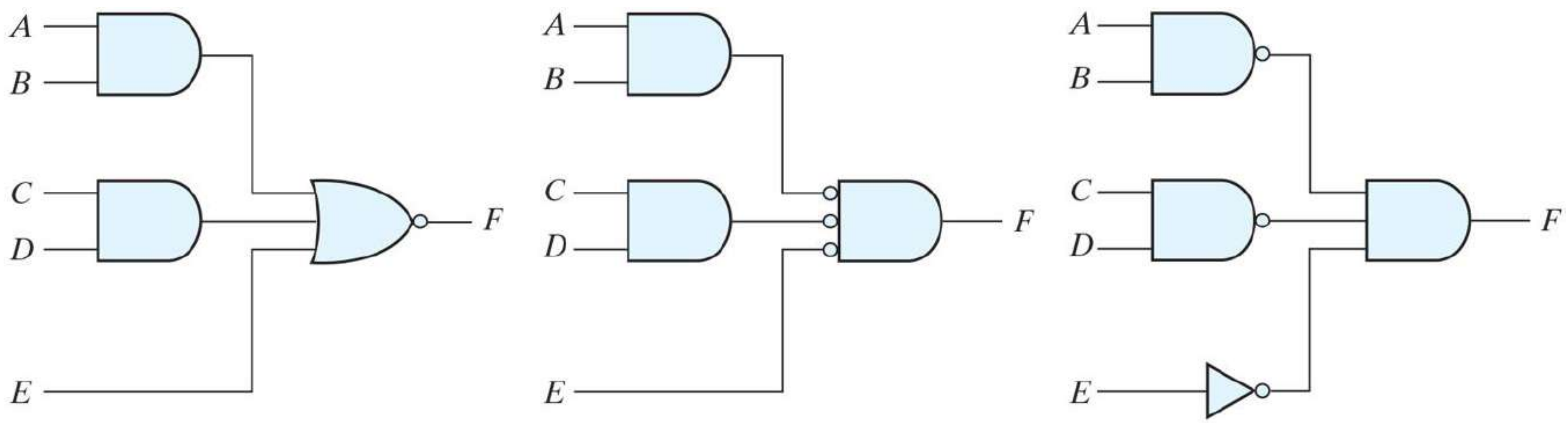
Nondegenerate Forms

- Considering four types of gates: AND, OR, NAND, and NOR
- We have 16 combination for first level – second level gates
  - Ex. AND-OR, AND-AND, NOR-OR
- Some combinations will ***degenerate*** to a single operation
  - Ex. AND-AND will degenerate to only AND operation
- Other combinations are ***nondegenerate*** (cannot be folded in a single operation), we have 8 such combinations
- First gate constitute the first level and the second gate is the second level in the implementation

AND–OR	OR–AND
NAND–NAND	NOR–NOR
NOR–OR	NAND–AND
OR–NAND	AND–NOR

AND–OR–INVERT Implementation

- NAND–AND and AND–NOR, are equivalent and can be treated Together
- Example $F = (AB + CD + E)$'.



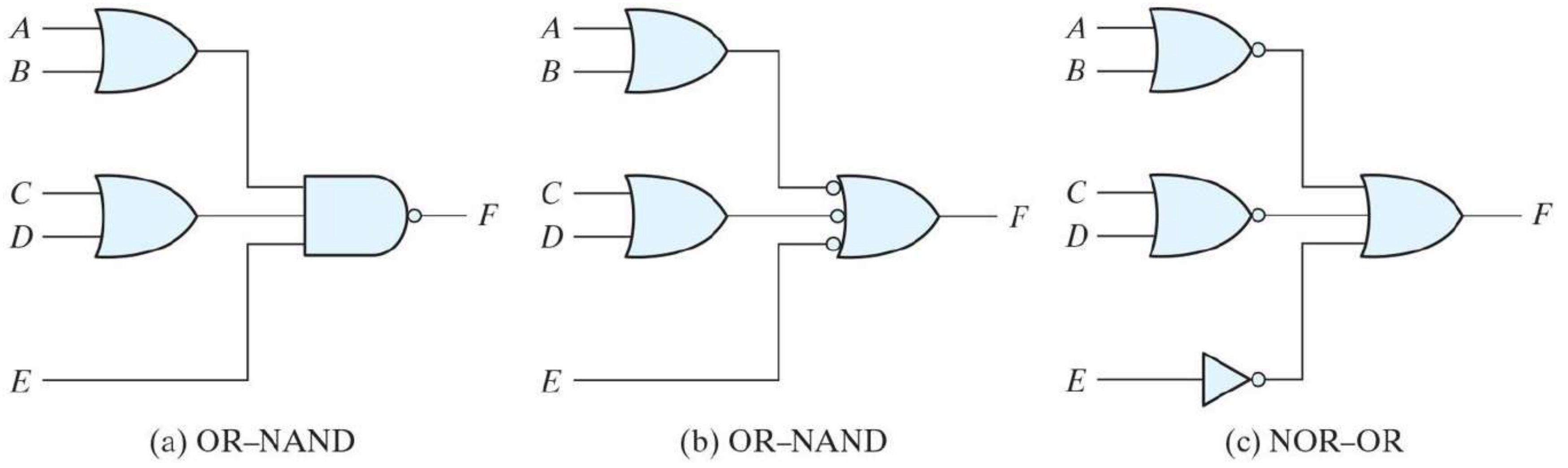
(a) AND–NOR (b) AND–NOR (c) NAND–AND

AND–OR–INVERT Implementation

- Like AND-OR implementation, AND-OR-INVERT requires Sum-of-Products form expression
- To implement a function F, we get the complement of the function $F^{`}$ SOP for the AND-OR-INVERT implementation

OR–AND–INVERT Implementation

- The OR–NAND and NOR–OR forms perform the OR–AND–INVERT function
- OR–NAND form resembles the OR–AND expect for inversion
  - Implementing $F = [(A + B)(C + D)E]`$



OR–AND–INVERT Implementation

- The OR–AND–INVERT implementation requires an expression in product-of-sums
- To implement a function F, we get the complement of the function $F^{`}$ POS for the OR-AND-INVERT implementation

Summary of Implementation with Other Two-Level Forms

- Because of the INVERT part in each case, we simplify the complement function $F`$ and use it in the implementation to get the function F

Table 3.2

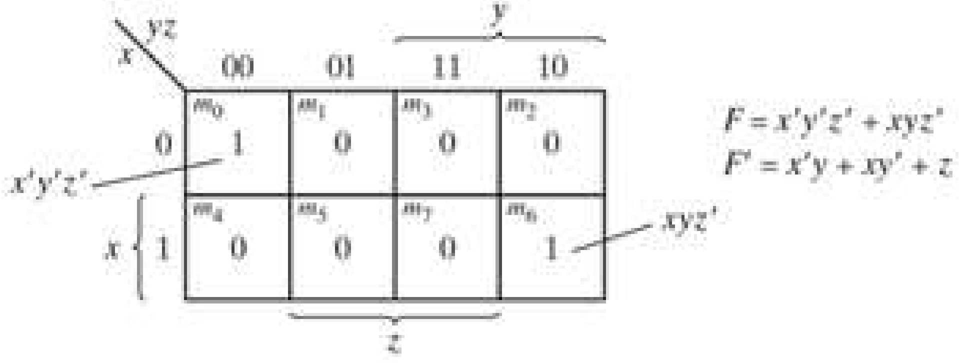
Equivalent Nondegenerate Implementation		Implements the Form	Simplify F' into	To Get an Output of
(a)	**(b)***			
AND–NOR	NAND–AND	AND–OR–INVERT	Sum-of-products form by combining 0's in the map.	F
OR–NAND	NOR–OR	OR–AND–INVERT	Product-of-sums form by combining 1's in the map and then complementing.	F

*Form (b) requires an inverter for a single literal term.

- Two-Level Implementations Example

Example 1

- Given the function
  - $F = x`y`z` + xyz`$
  - Implement it using the four combinations in the table 3.2
- First fill the K-map



(a) Map simplification in sum of products

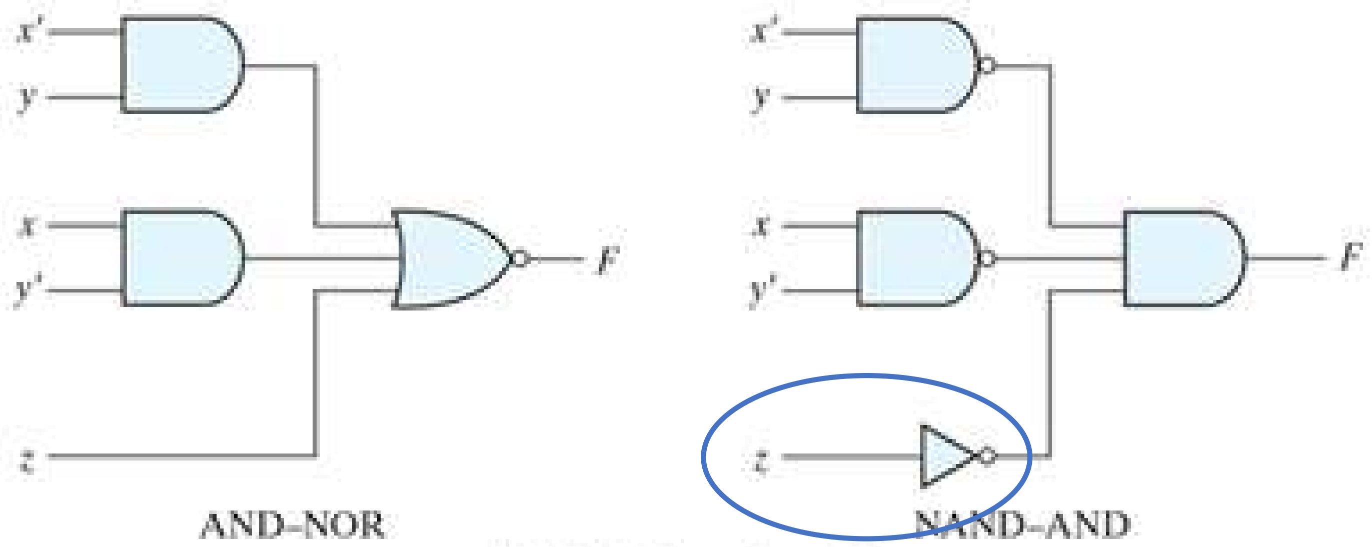
Example 1 (AND-OR-INVERT from)

- From the K-Map we can write the complement of the function
  - $F` = x`y + xy` + z$

- We can get the normal output function by inverting the expression
  - $F = (x`y + xy` + z)`$
  - This is the AND-OR-INVERT form
  - From this we can implement the AND-NOR, and NAND-AND (see next slide for the implementation)

Example 1 (AND-OR-INVERT form)

- Note that since Z is a single literal term it needs to be inverted in the NAND-AND implementation



(b) $F = (x'y + xy' + z)'$

Example 1 (OR–AND–INVERT form)

- OR-AND-INVERT needs complement of functions in product-of-sums form
  - First combine the 1’s in the K-Map then take the complement of the function (remember DeMorgan Theorem)
  - $F = x'y'z' + xyz'$
  - $F' = \overline{x'y'z' + xyz'}$ ➔ $F' = (\overline{x'y'z'})(\overline{xyz'})$ ➔
  - $F' = (\overline{x'} + \overline{y'} + \overline{z'})(\bar{x} + \bar{y} + \overline{z'})$ ➔
  - $F' = (x + y + z)(x' + y' + z)$

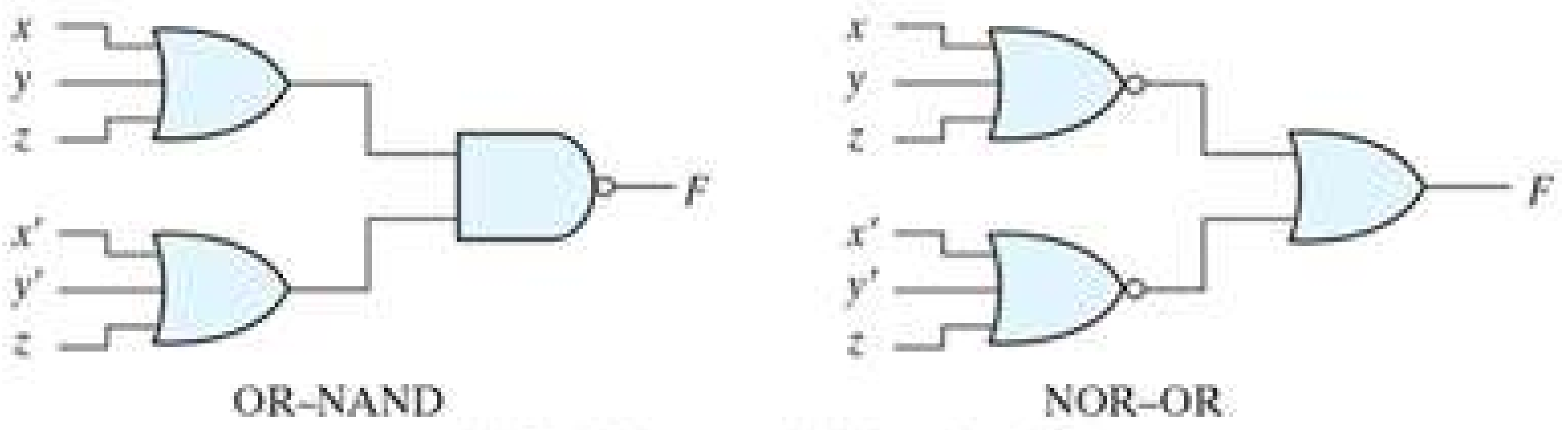
Example 1 (OR–AND–INVERT form)

- The normal form of the function becomes
- $$F = [(x + y + z)(x` + y` + z)]`$$
- This gives us the OR-AND-Inverted form

- From this we can implement the OR-NAND, and NOR-OR (see next slide for the implementation)

Example 1 (OR–AND–INVERT form)

-

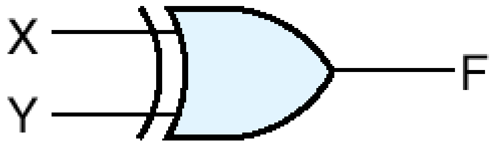


(c) $F = [(x + y + z)(x' + y' + z)]'$

- The *Exclusive OR* gate

Exclusive OR

- Exclusive OR (XOR) denoted by $\oplus$ is a logical operation perfuming
  - $x \oplus y = xy^{`} + x^{`}y$
- Exclusive NOR (XNOR) is the complement of XOR performing the operation
  - $(x \oplus y)^{`} = xy + x^{`}y^{`}$



XOR Gate Symbol

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

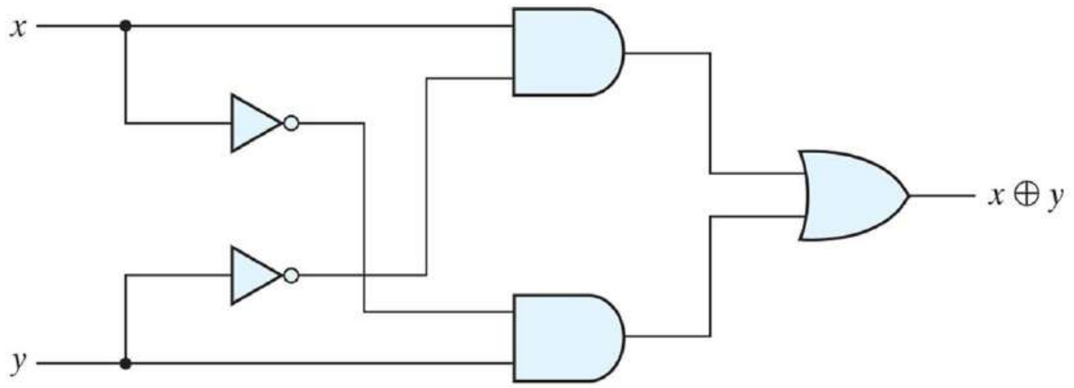
XOR Truth Table

XOR postulates and Theorems

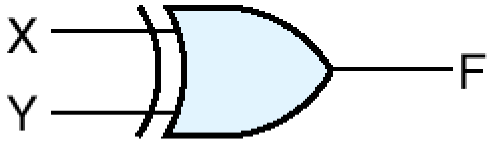
- Exclusive NOR (XNOR) can be generated by taking the complement of an XOR operation
  - $(x \oplus y)' = xy + x'y'$
- The following identities apply to XOR
  - $x \oplus 0 = x$
  - $x \oplus 1 = x'$
  - $x \oplus x = 0$
  - $x \oplus x' = 1$
  - $x \oplus y' = x' \oplus y = (x \oplus y)'$
- XOR is also commutative and associative

XOR Implementation

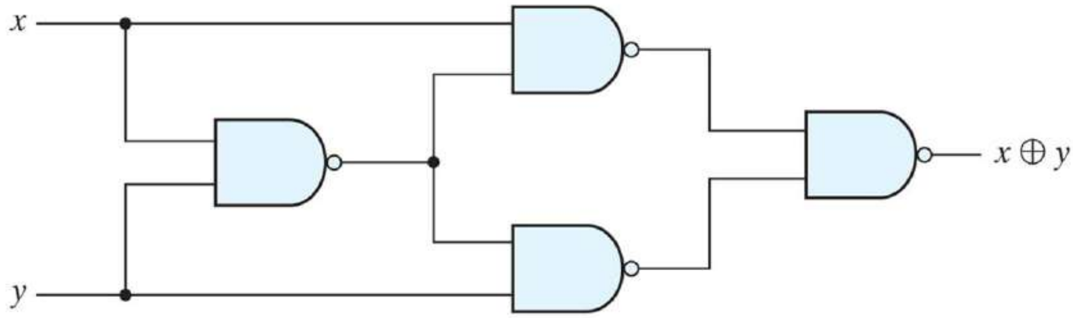
- XOR function can be implemented using AND-OR-NOT gates, or NAND gates only



(a) Exclusive-OR with AND–OR–NOT gates



X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0



(b) Exclusive-OR with NAND gates

XOR usage

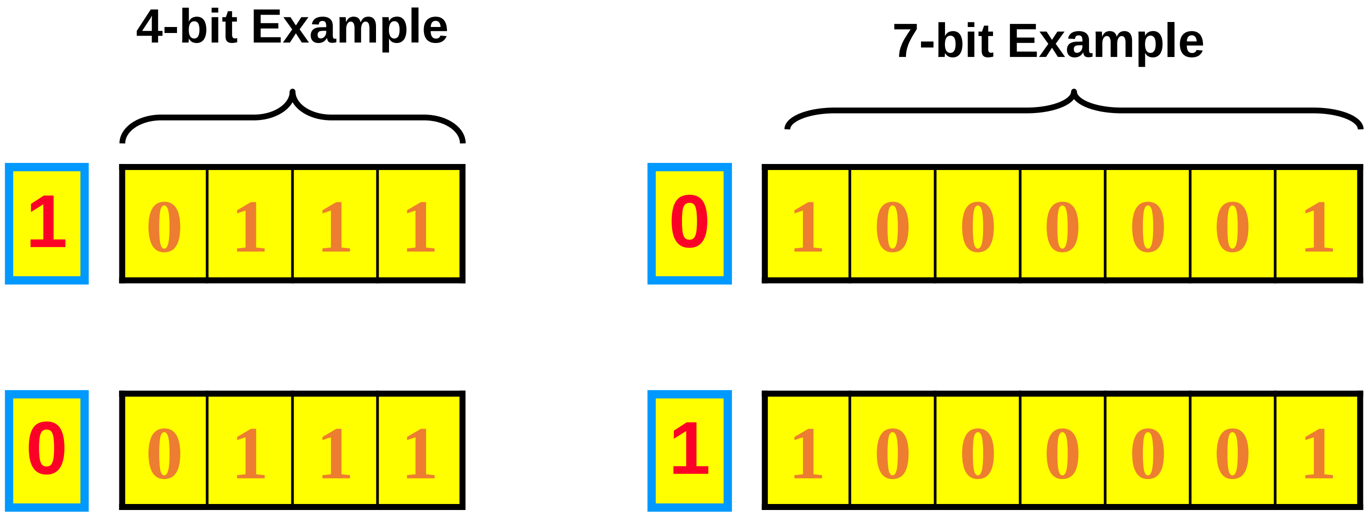
- Only few Boolean functions can be implemented/expressed using XOR gates
- This function is used a lot in in digital system design
- It is useful in arithmetic operations and error detection/correction

XOR = Odd Function

- The XOR operation with three or more variables can be converted into an ordinary Boolean function by replacing the $\oplus$ with its equivalent Boolean expression
  - $A \oplus B \oplus C = (AB' + A'B)C' + (AB + A'B')C$
  - $AB'C' + A'BC' + ABC + A'B'C$
  - $\sum(1, 2, 4, 7)$
- This function is equal to 1 only if one variable is equal to 1 or if all three variables are equal to 1.
  - This implies that an **odd number of variables must be one**. This is defined as an **odd function**.

Error Detecting Codes

- Parity
- One bit added to a group of bits to make the total number of ‘1’s (including the parity bit) even or odd
  - Even
  - Odd



- Good for checking single-bit errors

Parity Generation and Checking

- XOR functions are very useful in systems requiring error-detection and correction codes.
  - A circuit that generates a parity bit is called a parity generator.
  - The circuit that checks the parity is called a parity checker.
- Here is an example truth table of even-parity generator

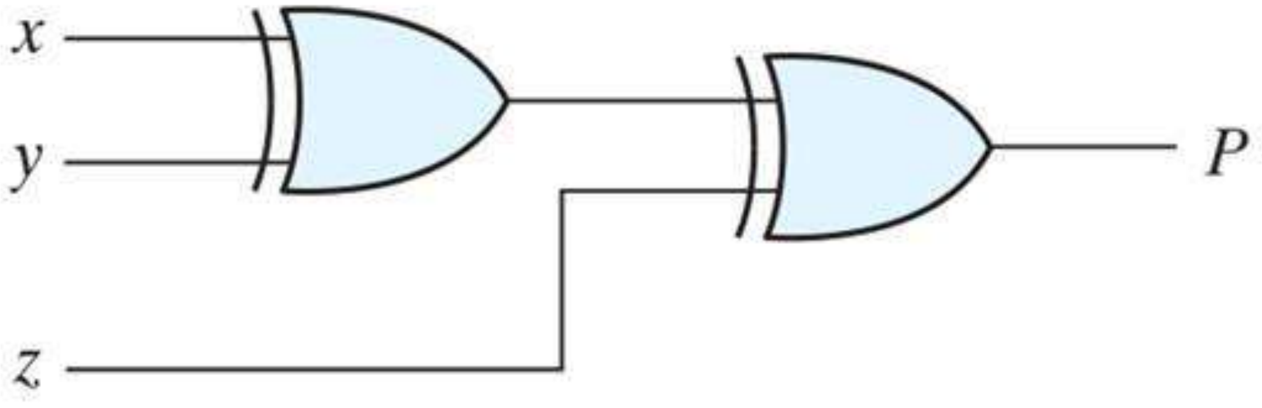
Three-Bit Message			Parity Bit
x	y	z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Parity Generator

- Design even parity generator for 3-bit signal
  - Perhaps make truth table and K-Map
  - Draw with XOR, then sum-of-products w/ NAND gates
- How do you design a detector?

3-Bit Even Parity Bit Generator Implementation

Three-Bit Message			Parity Bit
x	y	z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



(a) 3-bit even-parity generator

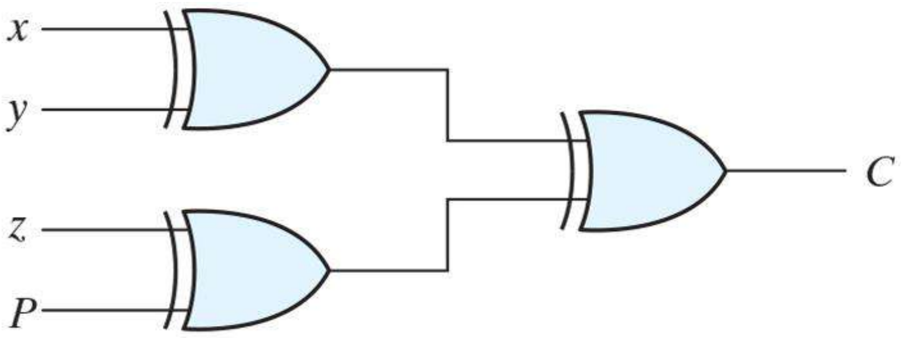
3-Bit Even Parity Bit Checker

- We receive the original 3 inputs + the parity bit
- If parity was violated the Error bit should be 1

Four Bits Received				Parity Error Check
x	y	z	P	C
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

3-Bit Even Parity Bit Checker Implementation

Four Bits Received				Parity Error Check
x	*y*	*z*	*P*	*C*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0



(b) 4-bit even-parity checker

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 7

Combinational Logic

CONTENTS

- Combinational Circuits
- Analysis Procedure
- Design Procedure

Weekly Learning Outcomes

1. Describe Combinational Circuits
2. Design Combinational Circuits
3. Given its logic diagram, know how to analyze a combinational logic circuit

Required Reading

1. Chapter 3 (4.1 to 4.4)
(Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapter 6

(Digital Logic for Computing) (John Seiffertt Digital Logic for Computing)

Introduction

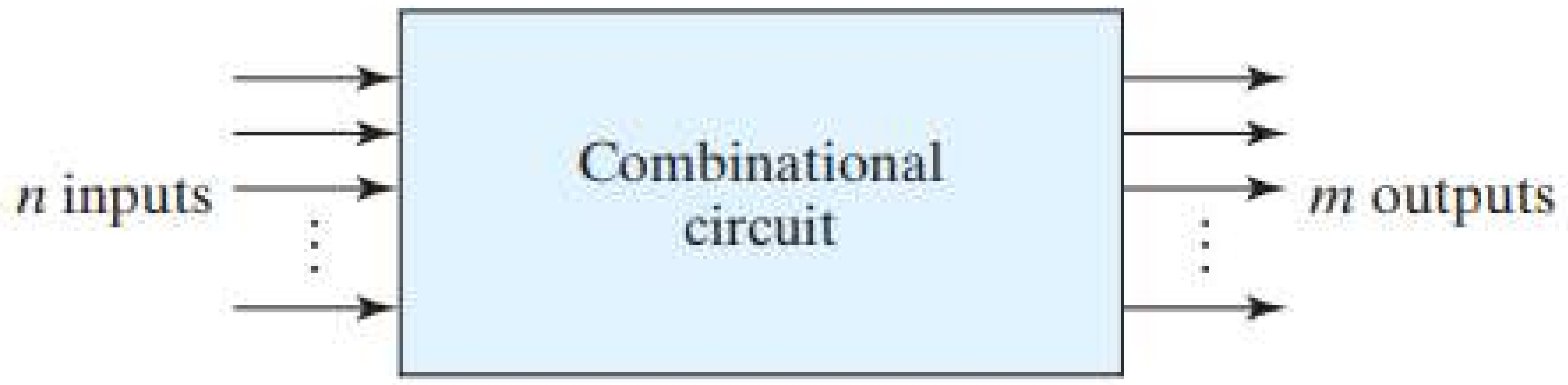
Combinational Circuits

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs.
- A combinational circuit performs an operation that can be specified logically by a set of Boolean functions.

Combinational Circuits

A combinational circuit consists of an interconnection of logic gates. Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data.

Block Diagram of Combinational Logic



The *n* input binary variables come from an external source; the *m* output variables are produced by the internal combinational logic circuit and go to an external destination.

Combinational Logic

Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0.

**Note*: Logic simulators show only 0’s and 1’s, not the actual analog signals.

Combinational Logic

- For *n* input variables, there are 2*n* possible combinations of the binary inputs. For each possible input combination, there is one possible value for each output variable. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables.
- A combinational circuit also can be described by *m* Boolean functions, one for each output variable. Each output function is expressed in terms of the *n* input variables.

Analysis Procedure

Analysis Procedure

- If the logic diagram to be analyzed is accompanied by a function name or an explanation of what it is assumed to accomplish, then the analysis problem reduces to a verification of the stated function.
- The analysis can be performed manually by finding the Boolean functions or truth table or by using a computer simulation program.

Analysis Procedure

- The first step in the analysis is to make sure that the given circuit is combinational and not sequential. **The diagram of a combinational circuit has logic gates with no feedback paths** or **memory elements .**
- A feedback path is a connection from the output of one gate to the input of a second gate whose output forms part of the input to the first gate. Feedback paths in a digital circuit define a sequential circuit

Analysis Procedure

Once the logic diagram is verified to be that of a combinational circuit, one can proceed to obtain the output Boolean functions or the truth table. If the function of the circuit is under investigation, then it is necessary to interpret the operation of the circuit from the derived Boolean functions or truth table.

Analysis Procedure

To obtain the output Boolean functions from a logic diagram, we proceed as follows:

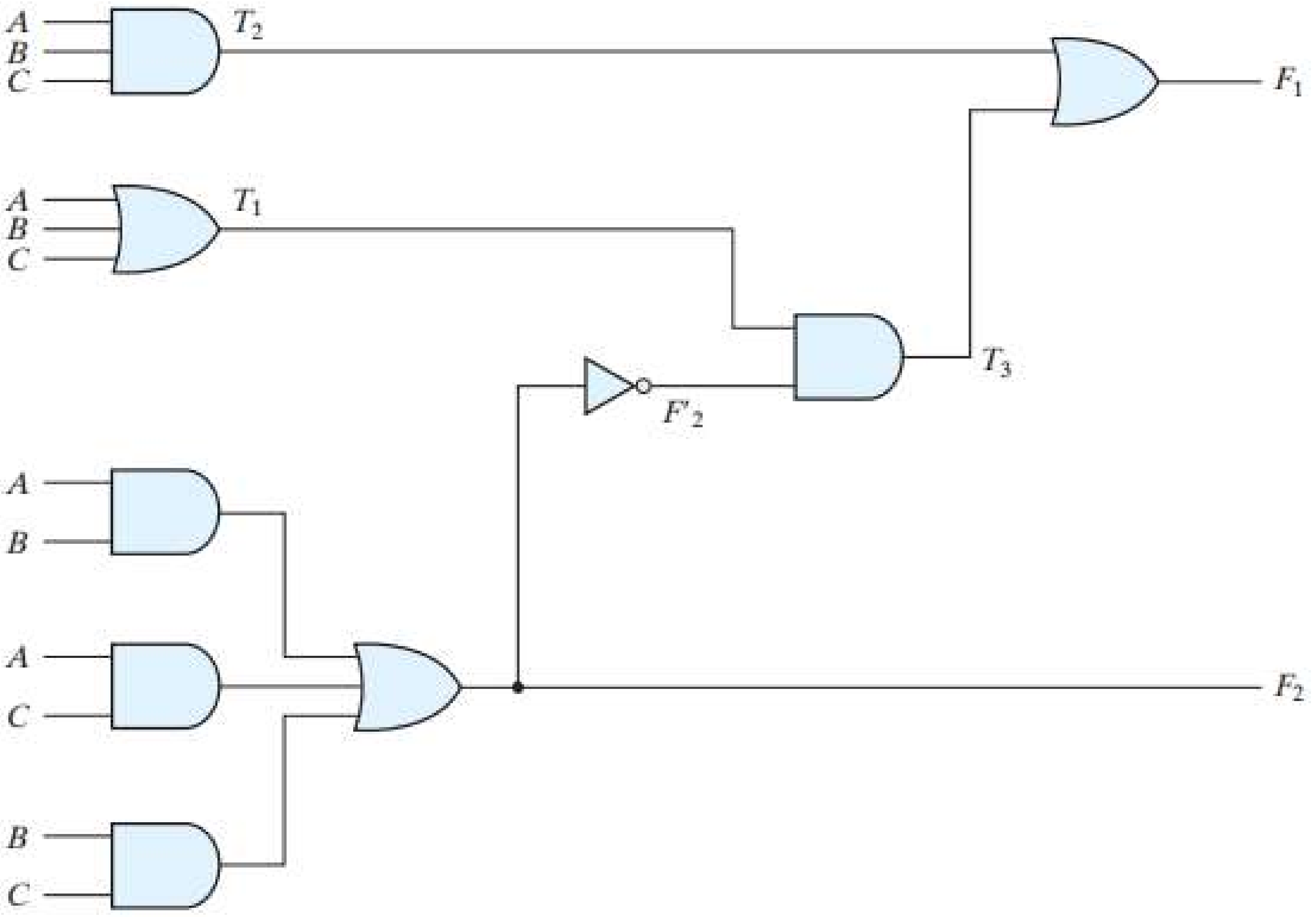
1. Label all gate outputs that are a function of input variables with arbitrary symbols—but with meaningful names. Determine the Boolean functions for each gate output.

2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.

3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.

4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

Analysis Procedure - Example



Analysis Procedure - Example

The circuit has three binary inputs—*A*, *B*, and *C*—and two binary outputs—*F*1 and *F*2. The outputs of various gates are labeled with intermediate symbols.

The outputs of gates that are a function only of input variables are *T*1 and *T*2.

$$F_2 = AB + AC + BC$$
$$T_1 = A + B + C$$
$$T_2 = ABC$$

Analysis Procedure - Example

Next, we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F'_2 T_1$$
$$F_1 = T_3 + T_2$$

To obtain *F*1 as a function of *A*, *B*, and *C*, we form a series of substitutions as follows:

Analysis Procedure - Example

$$
\begin{aligned}
F_1 &= T_3 + T_2 = F'_2T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\
&= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\
&= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\
&= A'BC' + A'B'C + AB'C' + ABC
\end{aligned}
$$

Analysis Procedure

Merely finding a Boolean representation of a circuit doesn't provide insight into its behavior, but in this example we will observe that the Boolean equations and truth table for $F1$ and $F2$

To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, we proceed as follows:

Analysis Procedure

1. Determine the number of input variables in the circuit. For *n* inputs, form the 2^n possible input combinations and list the binary numbers from $0 \text{ to } (2^n - 1)$ in a table.

2. Label the outputs of selected gates with arbitrary symbols.

3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.

4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

Analysis Procedure

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design Procedure

Design Procedure

The design of combinational circuits starts from the specification of the design objective and culminates in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained. The procedure involves the following steps:

Design Procedure

1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.

2. Derive the truth table that defines the required relationship between inputs and outputs.

3. Obtain the simplified Boolean functions for each output as a function of the input variables.

4. Draw the logic diagram and verify the correctness of the design (manually or by simulation).

Design Procedure - Code Conversion Example

- To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B.
- A combinational circuit performs this transformation by means of logic gates. The design procedure will be illustrated by an example that converts binary coded decimal (BCD) to the excess-3 code for the decimal digits.

Design Procedure - Code Conversion Example

The bit combinations assigned to the BCD and excess-3 codes are listed in the Table

Decimal Digit	BCD 8421	2421	Excess-3	8, 4, −2, −1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
	1010	0101	0000	0001
Unused	1011	0110	0001	0010
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1100
nations	1110	1001	1110	1101
	1111	1010	1111	1110

Design Procedure - Code Conversion Example

Since each code uses four bits to represent a decimal digit, there must be four input variables and four output variables.

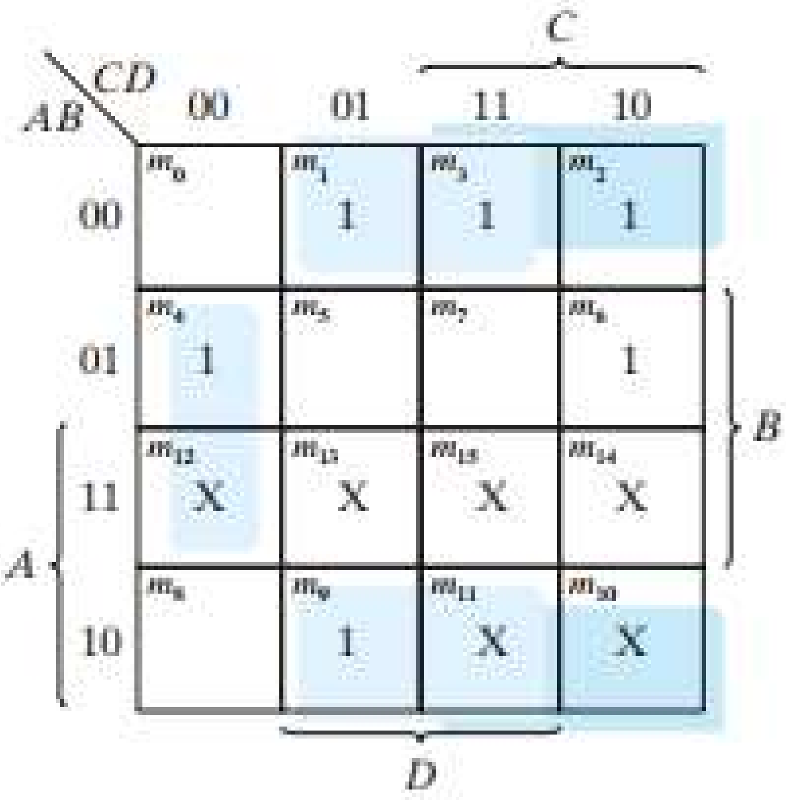
Truth Table for Code Conversion Example

Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

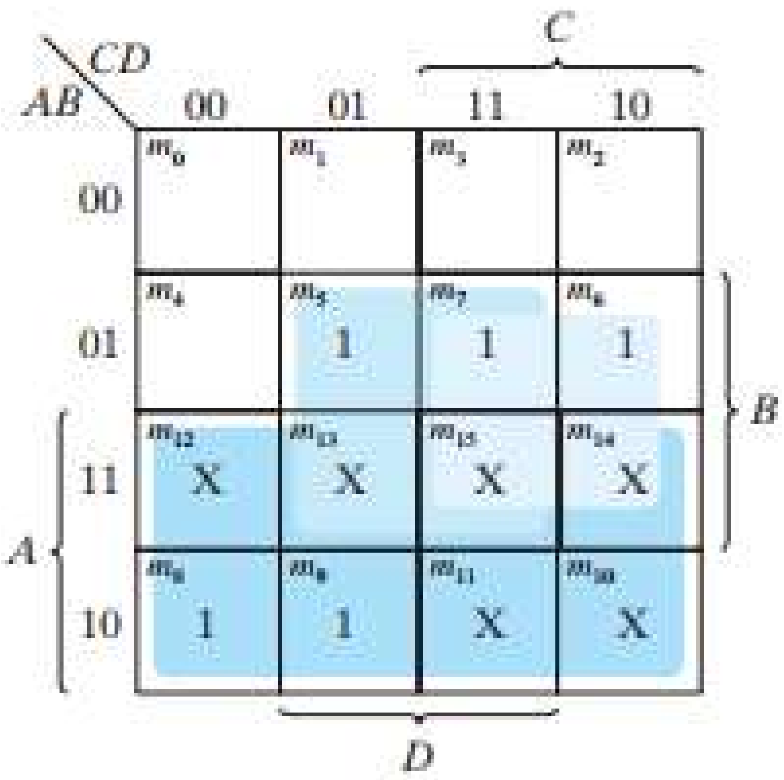
Design Procedure - Code Conversion Example

Note that four binary variables may have 16 bit combinations, but only 10 are listed in the truth table. The six bit combinations not listed for the input variables are don’t-care combinations. These values have no meaning in BCD and we assume that they will never occur in actual operation of the circuit.

Design Procedure - Code Conversion Example

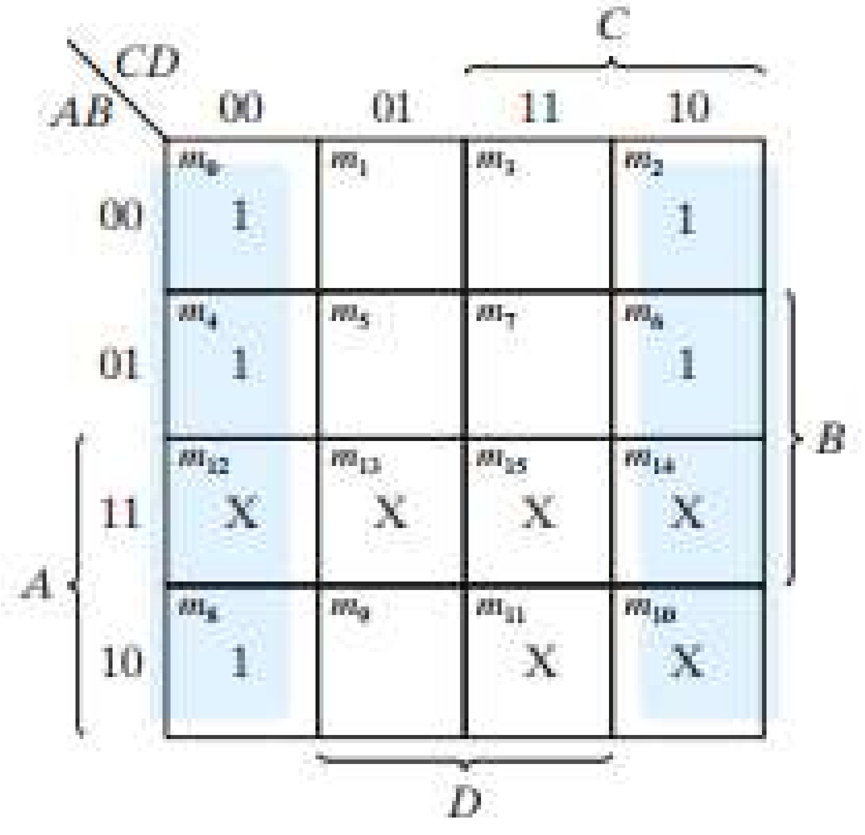


$x = B'C + B'D + BC'D'$

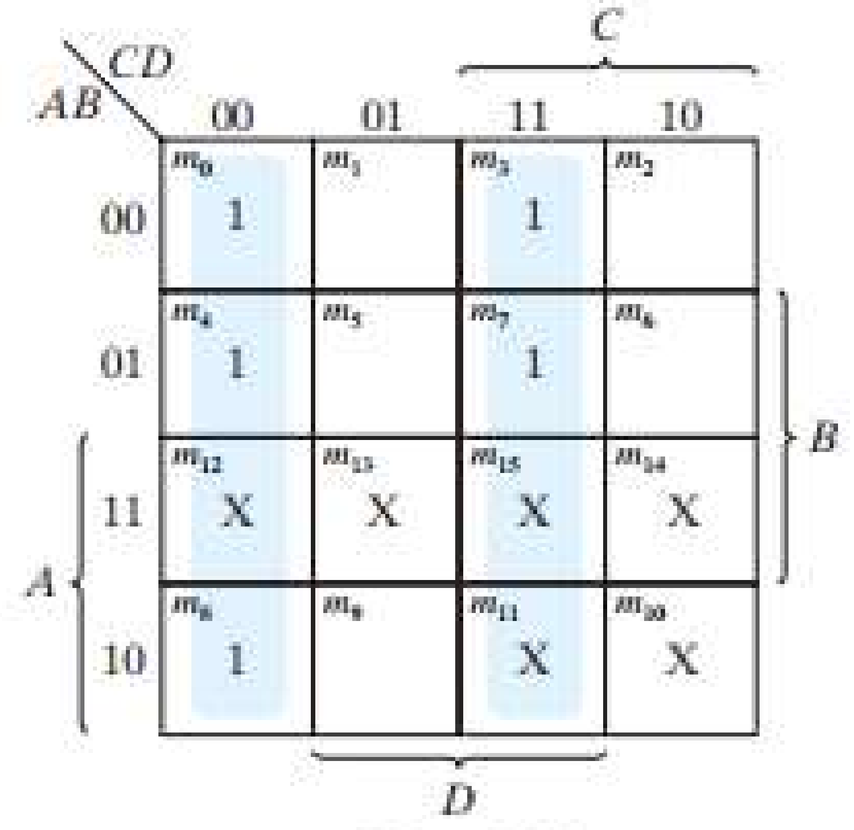


$w = A + BC + BD$

Design Procedure - Code Conversion Example



$z = D'$



$y = CD + C'D'$

Design Procedure - Code Conversion Example

Each one of the four maps represents one of the four outputs of the circuit as a function of the four input variables. The 1's marked inside the squares are obtained from the minterms that make the output equal to 1. The 1's are obtained from the truth table by going over the output columns one at a time.

Design Procedure - Code Conversion Example

For example, the column under output *z* has five 1's; therefore, the map for *z* has five 1's, each being in a square corresponding to the minterm that makes *z* equal to 1. The six don't-care minterms 10 through 15 are marked with an *X*.

Design Procedure - Code Conversion Example

$$z = D'$$

$$y = CD + C'D' = CD + (C + D)'$$

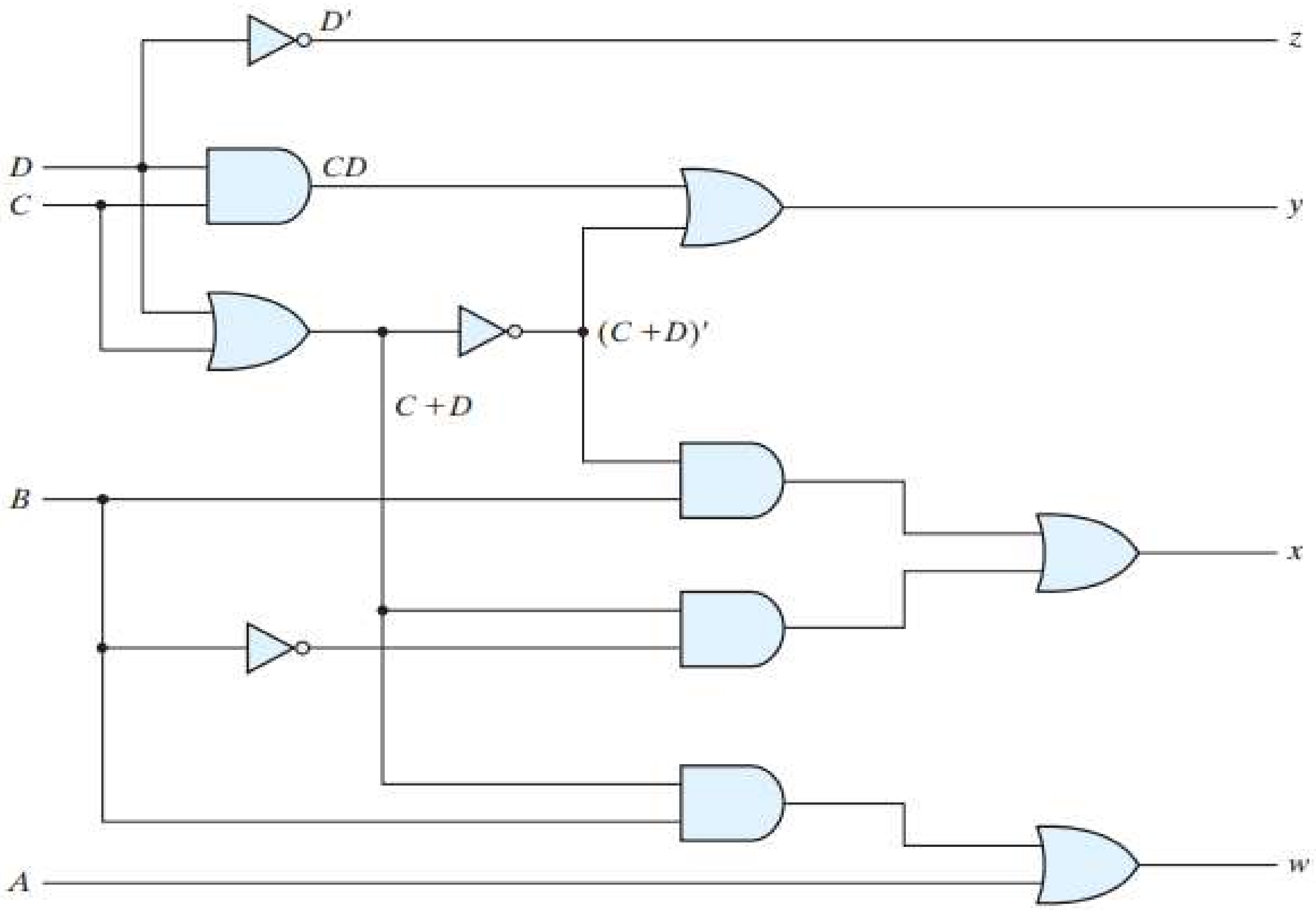
$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$

$$= B'(C + D) + B(C + D)'$$

$$w = A + BC + BD = A + B(C + D)$$

The logic diagram that implements these expressions is shown in next slide.

Design Procedure - Code Conversion Example



Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 9
Combinational Logic

CONTENTS

Weekly Learning Outcomes

1. Design Binary Adder-Subtractor
2. Understand Decimal Adder
3. Describe the functionality of Binary Multiplier and Magnitude Comparator

Required Reading

1. Chapter 4 (4.5 to 4.8)
(Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapter 7

(Digital Logic for Computing) (John Seiffertt Digital Logic for Computing)

Binary Adder-Subtractor

Binary Adder-Subtractor

Digital computers perform a variety of information-processing tasks. Among the functions encountered are the various arithmetic operations. The most basic arithmetic operation is the addition of two binary digits.

This simple addition consists of four possible elementary operations: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10. The first three operations produce a sum of one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a *carry*.

Binary Adder-Subtractor

Half Adder

When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher order pair of significant bits. A combinational circuit that performs the addition of two bits is called a *half adder*.

Full Adder

One that performs the addition of three bits (two significant bits and a previous carry) is a *full adder*.

Binary Adder-Subtractor

Binary adder–subtractor

A binary adder–subtractor is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers.

Half Adder

- Circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry.
- We assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs. The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum.

Half Adder – Boolean Function

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

$$S = x'y + xy'$$
$$C = xy$$

The truth table for the half adder is listed in next slide.

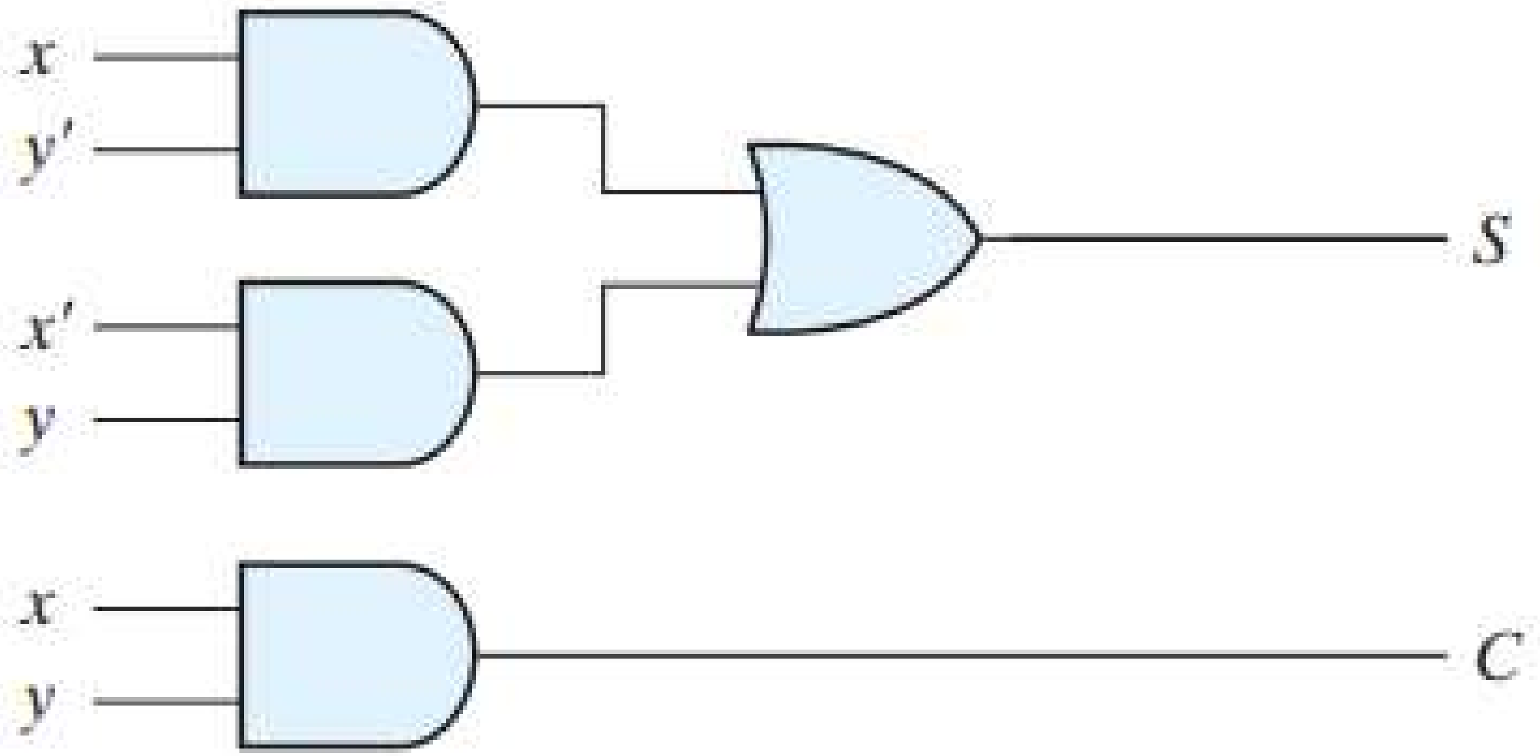
Half Adder – Truth Table

Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth Table for Half Adder

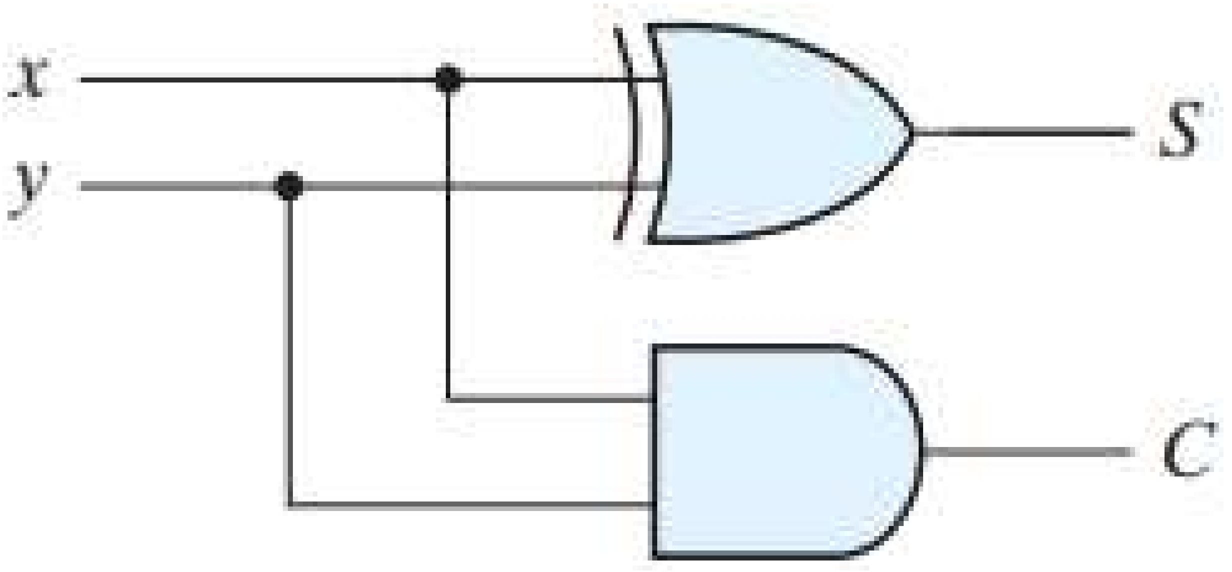
Half Adder – Logic Diagram



(a) $S = xy' + x'y$
$C = xy$

Half Adder – Logic Diagram

It can be also implemented with an exclusive-OR and an AND gate as shown below.



(b) $S = x \oplus y$
$C = xy$

Full Adder

- A full adder is a combinational circuit that forms the arithmetic sum of three bits.
- It consists of three inputs and two outputs. Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous lower significant position.
- Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary representation of 2 or 3 needs two bits.

Full Adder

- The two outputs are designated by the symbols *S* for sum and *C* for carry.
- The binary variable *S* gives the value of the least significant bit of the sum.
- The binary variable *C* gives the output carry formed by adding the input carry and the bits of the words.

Full Adder - Boolean Function

When all input bits are 0, the output is 0. The *S* output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The *C* output has a carry of 1 if two or three inputs are equal to 1.

The simplified expressions are

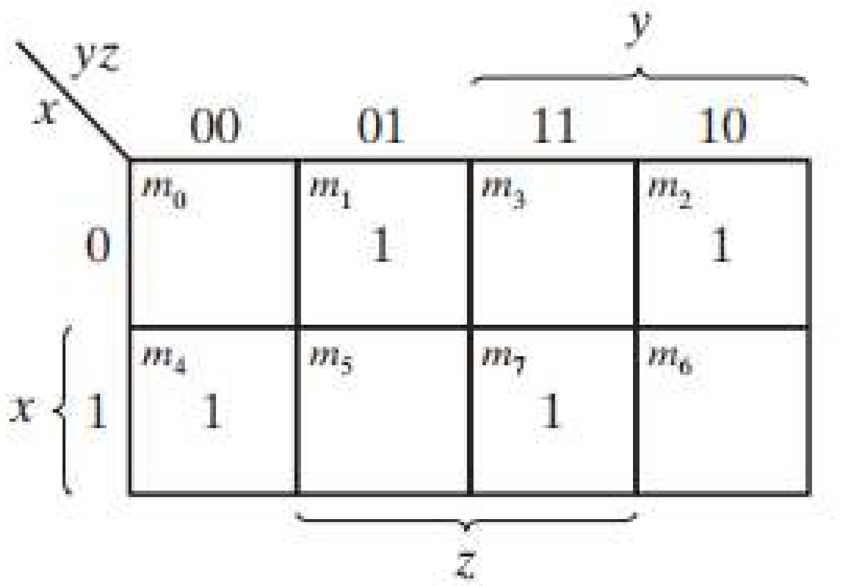
$$S = x'y'z + x'yz' + xy'z' + xyz$$
$$C = xy + xz + yz$$

Full Adder - Truth Table

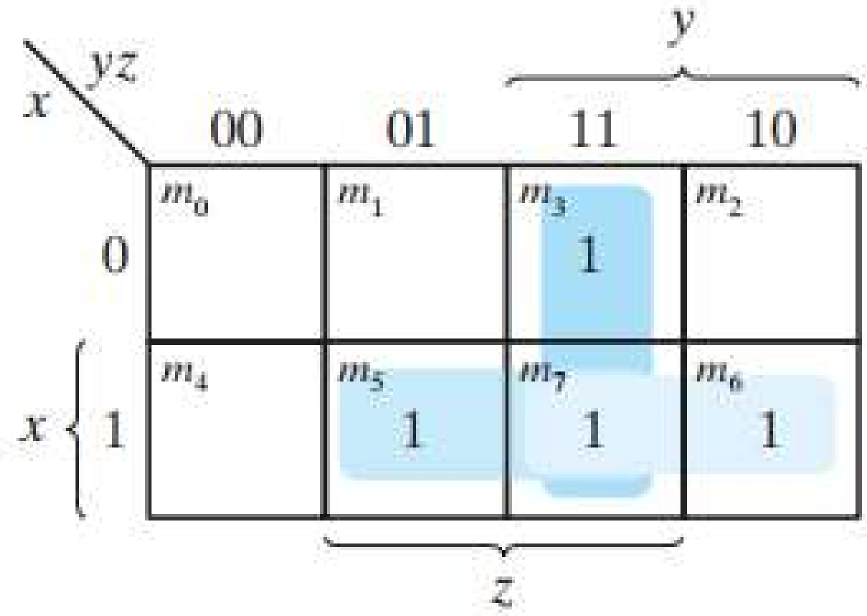
Full Adder

x	*y*	*z*	*C*	*S*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder - Map

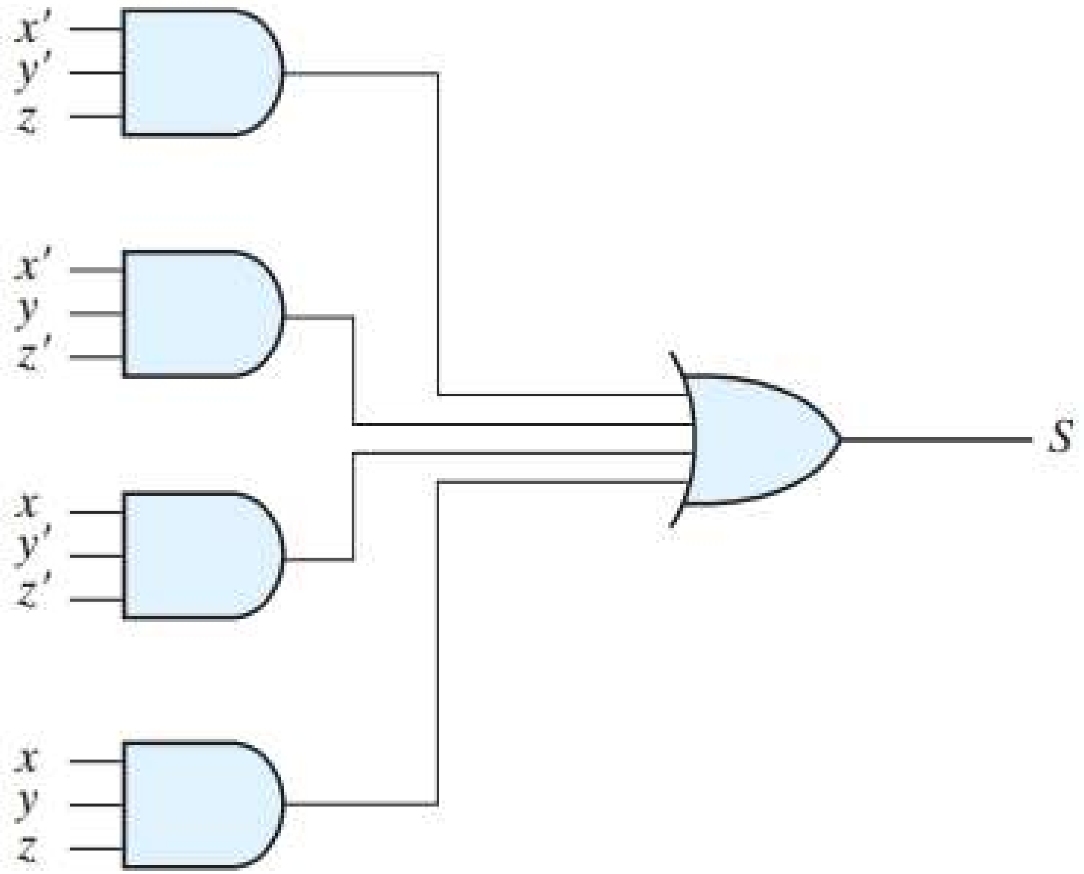


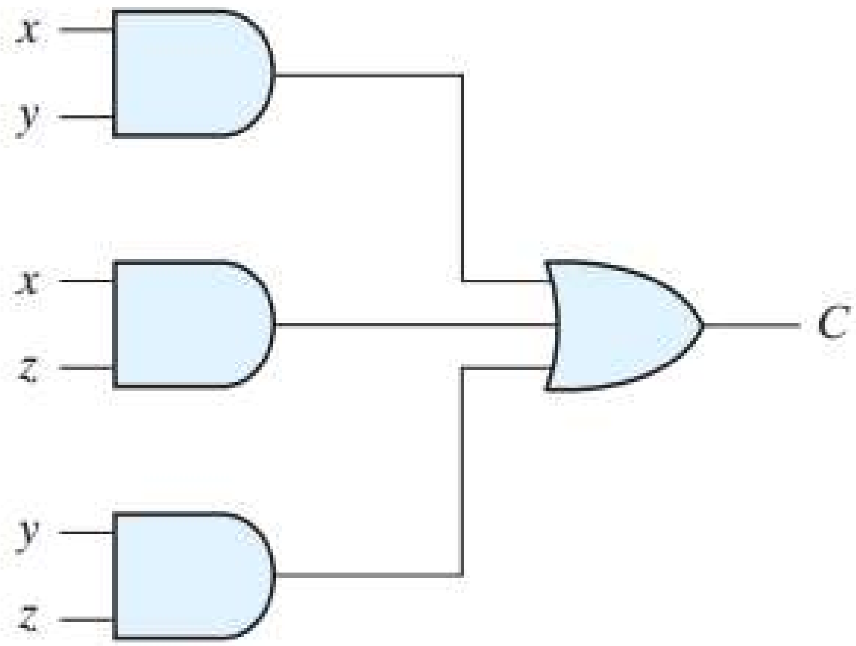
(a) $S = x'y'z + x'yz' + xy'z' + xyz$



(b) $C = xy + xz + yz$

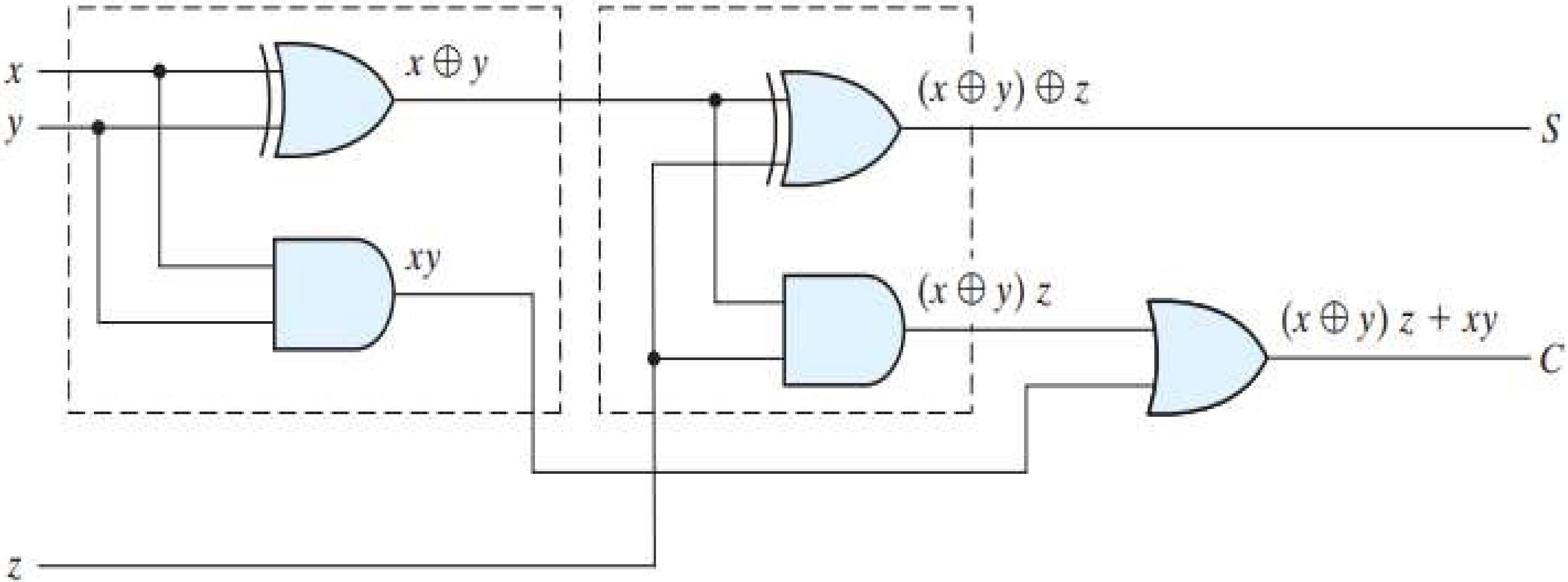
Full Adder - Logic Diagram (Sum of Product)





Full Adder - Logic Diagram

It can also be implemented with two half adders and one OR gate



Full Adder - Logic Diagram

The *S* output from the second half adder is the exclusive-OR of *z* and the output of the first half adder, giving

$$
\begin{aligned}
S &= z \oplus (x \oplus y) \\
&= z'(xy' + x'y) + z(xy' + x'y)' \\
&= z'(xy' + x'y) + z(xy + x'y') \\
&= xy'z' + x'yz' + xyz + x'y'z
\end{aligned}
$$

The carry output is

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

Binary Adder

- A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers.
- It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain.

Binary Adder - Implementation

Binary Adder can be implemented with

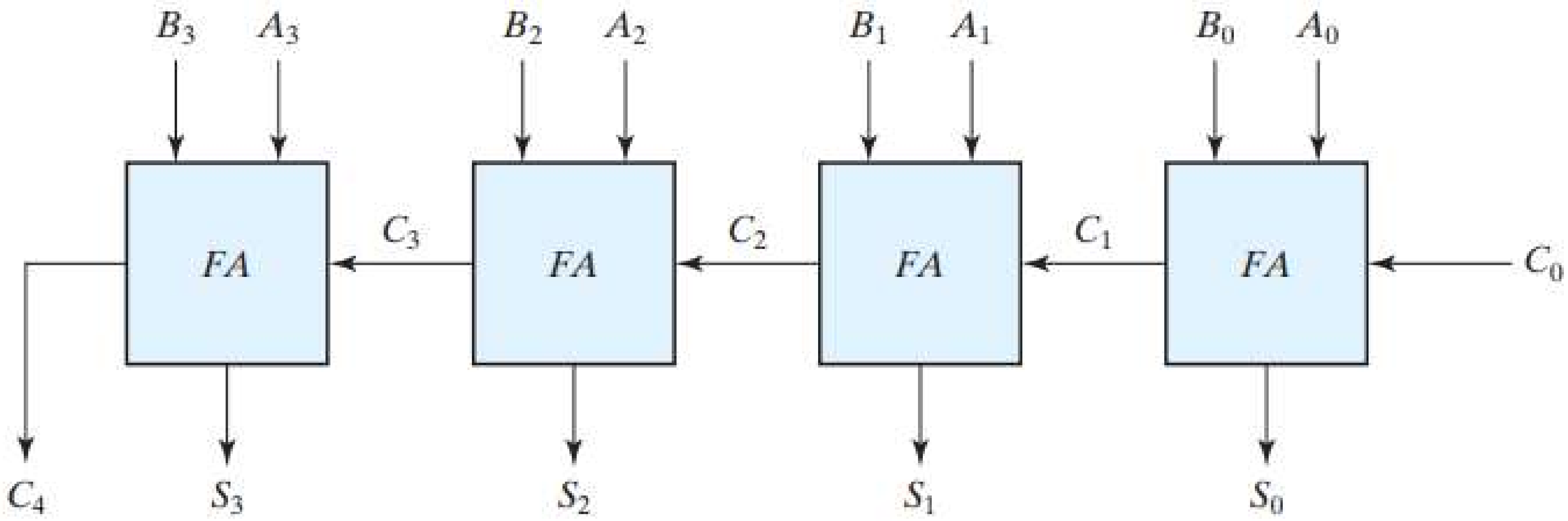
1. Chain of n full adders or

2. Chain of one-half adder and (n – 1) full adders

Binary Adder - Example

- Interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder is shown in next slide.
- The augend bits of *A* and the addend bits of *B* are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit.
- The carries are connected in a chain through the full adders.
- The input carry to the adder is C_0, and it ripples through the full adders to the output carry C_4. The *S* outputs generate the required sum bits.

Binary Adder - Four-bit adder



Binary Adder - Four-bit adder

Consider the two binary numbers A = 1011 and B = 0011. Their sum S = 1110 is formed with the four-bit adder as follows:

Subscript i:	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

Binary Adder - Four-bit adder working

- The bits are added with full adders, starting from the least significant position (subscript 0), to form the sum bit and carry bit.
- The input carry C_0 in the least significant position must be 0. The value of C_{i+1} in a given significant position is the output carry of the full adder. This value is transferred into the input carry of the full adder that adds the bits one higher significant position to the left.
- The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated.

Propagation Delay

- The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit.
- The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders.
- The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added.

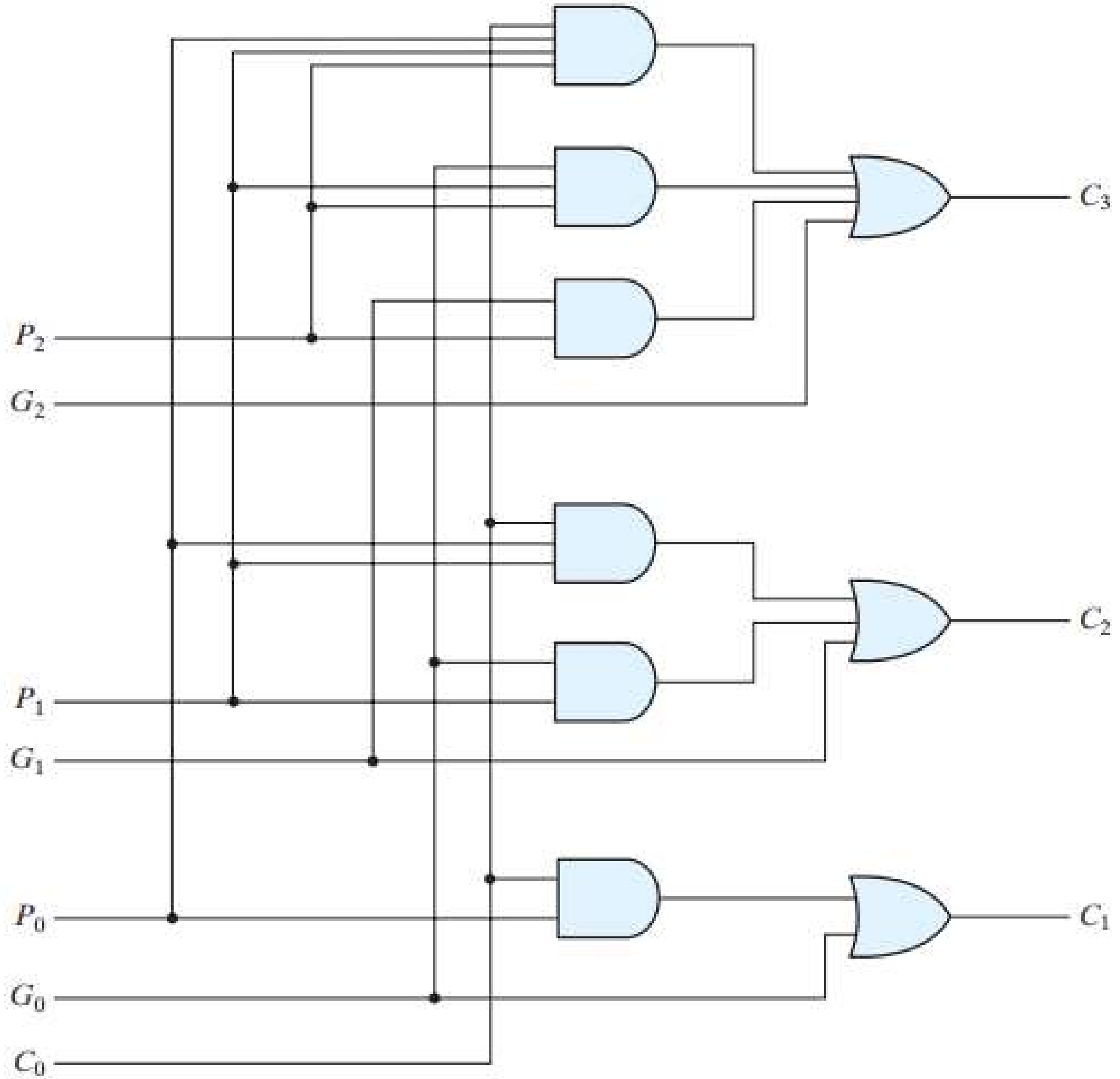
Propagation Delay

1. An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays. However, physical circuits have a limit to their capability.
2. Another solution is to increase the complexity of the equipment in such a way that the carry delay time is reduced.

There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of *carry lookahead logic*.

Propagation Delay

Consider the circuit of the full adder shown below



Propagation Delay

If we define two new binary variables

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

Gi is called a *carry generate*, and it produces a carry of 1 when both *Ai* and *Bi* are 1, regardless of the input carry *Ci*. *Pi* is called a *carry propagate.*

Propagation Delay

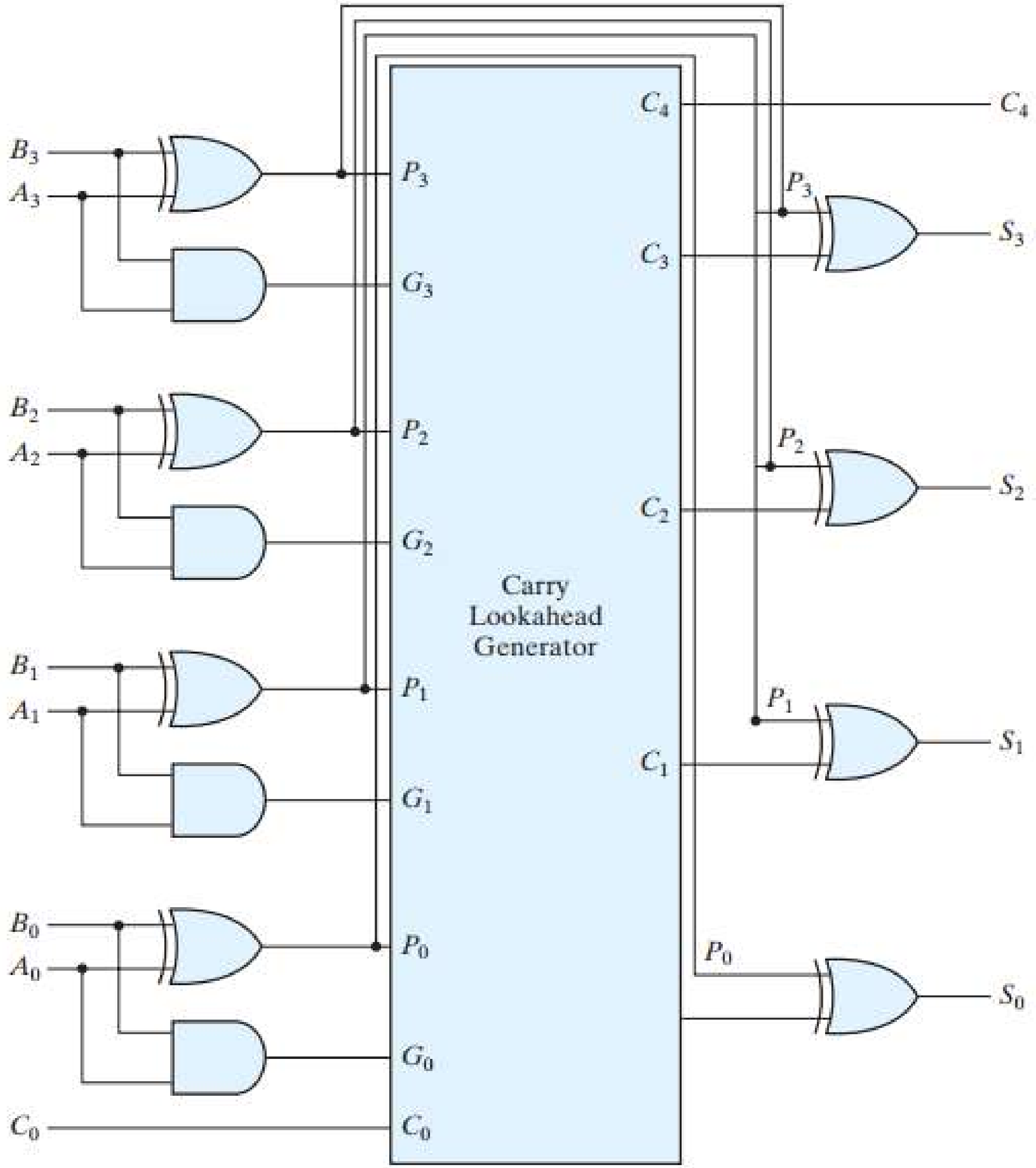
Boolean functions for the carry outputs of each stage and substitute the value of each *Ci* from the previous equations

$$C_0 = \text{input carry}$$
$$C_1 = G_0 + P_0C_0$$

$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 = P_2P_1P_0C_0$$

Four-bit adder with a carry lookahead scheme is shown in next slide.

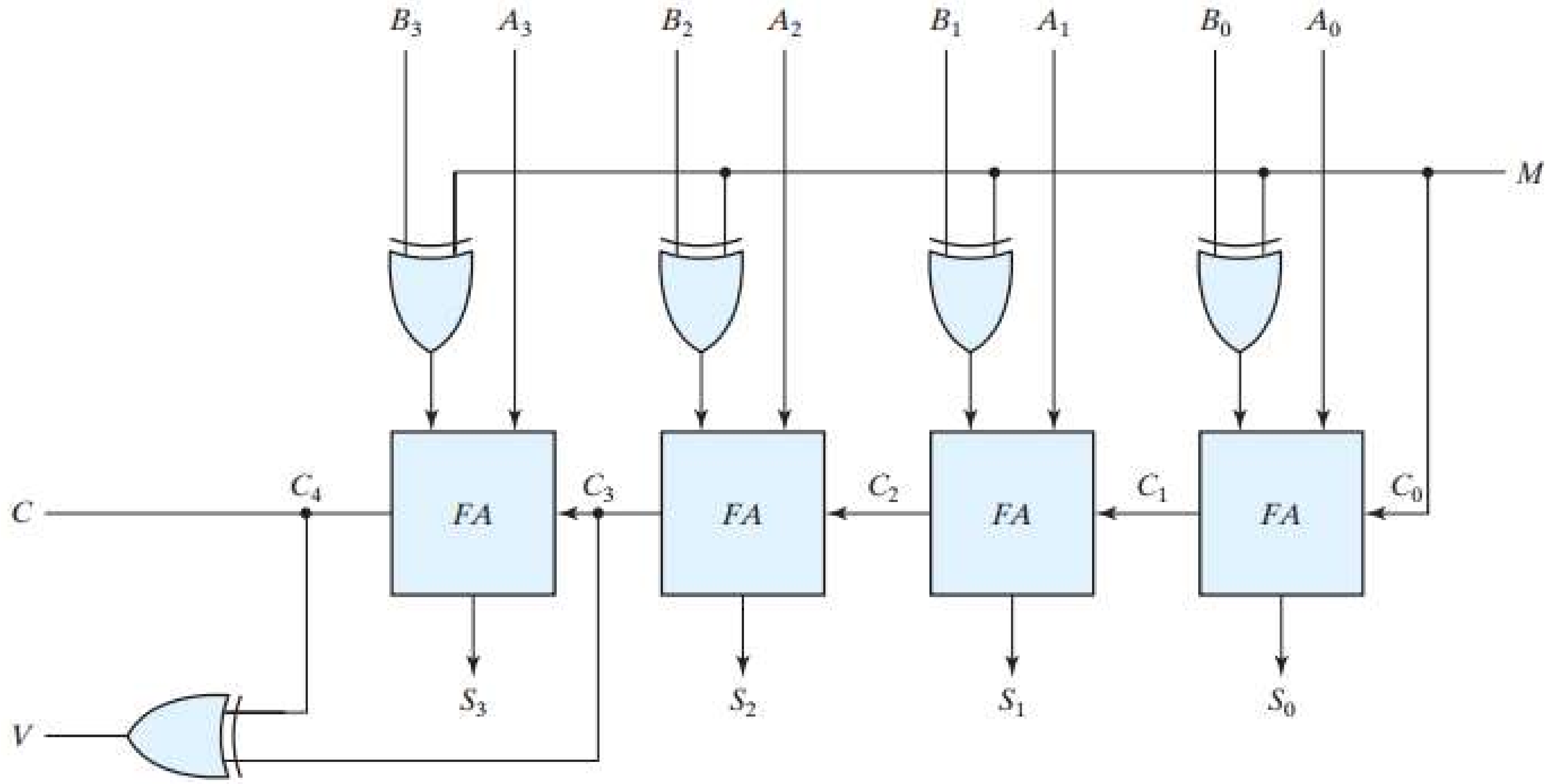
Propagation Delay



Binary Subtractor

- The subtraction of unsigned binary numbers can be done most conveniently by means of complements.
- Subtraction *A* - *B* can be done by taking the 2's complement of *B* and adding it to *A*. The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.
- The addition and subtraction operations can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder.

Four Bit Adder-Subtractor



Decimal Adder

Decimal Adder – BCD Adder

- Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input carry.
- Suppose we apply two BCD digits to a four-bit binary adder. The adder will form the sum in *binary* and produce a result that ranges from 0 through 19.

Decimal Adder – BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

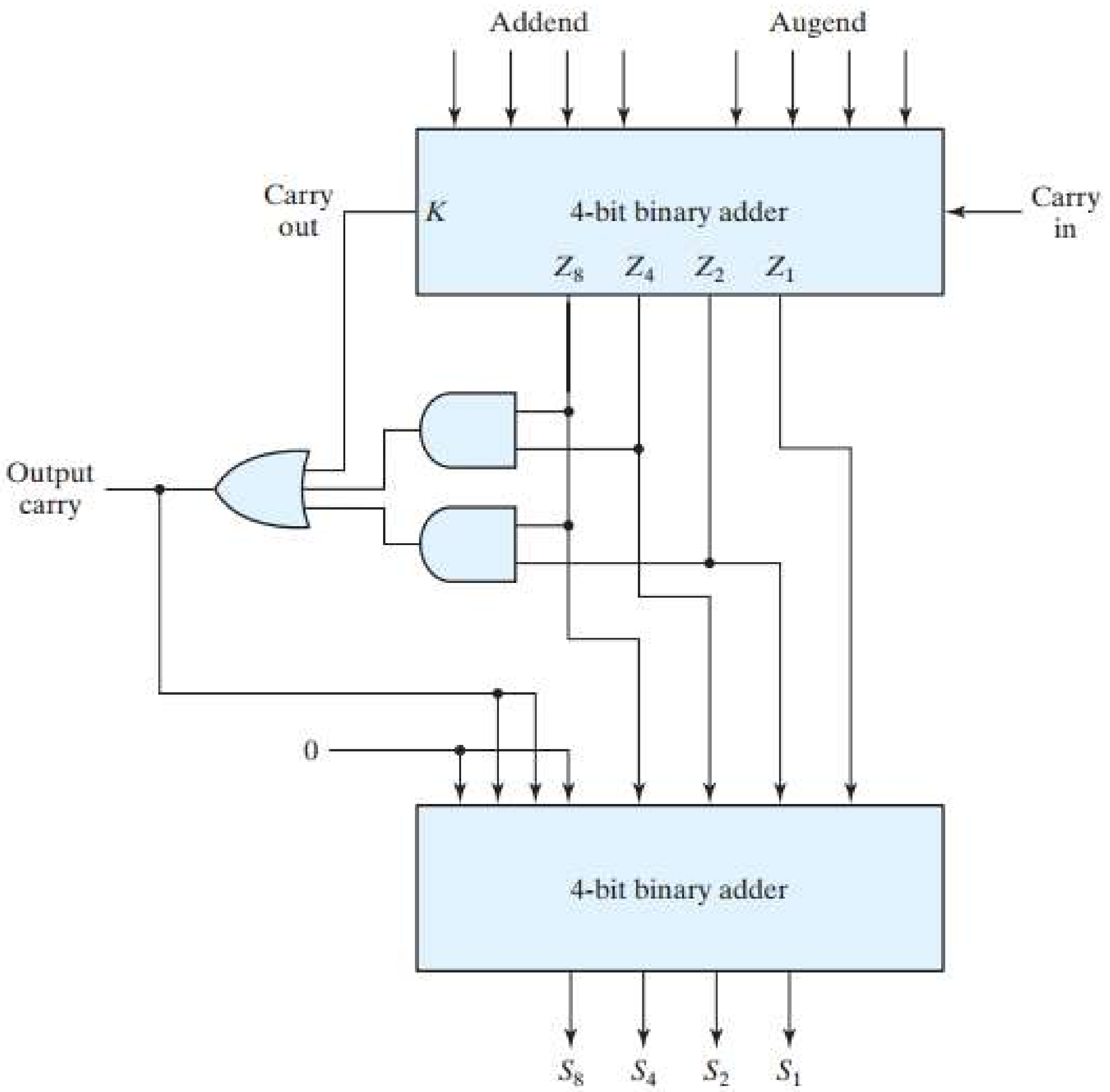
Decimal Adder – BCD Adder

The problem is to find a rule by which the binary sum is converted to the correct BCD digit representation of the number in the BCD sum. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

When *C* = 1, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage

Decimal Adder – BCD Adder (Block Diagram)



Binary Multiplier

Binary Multiplier - Introduction

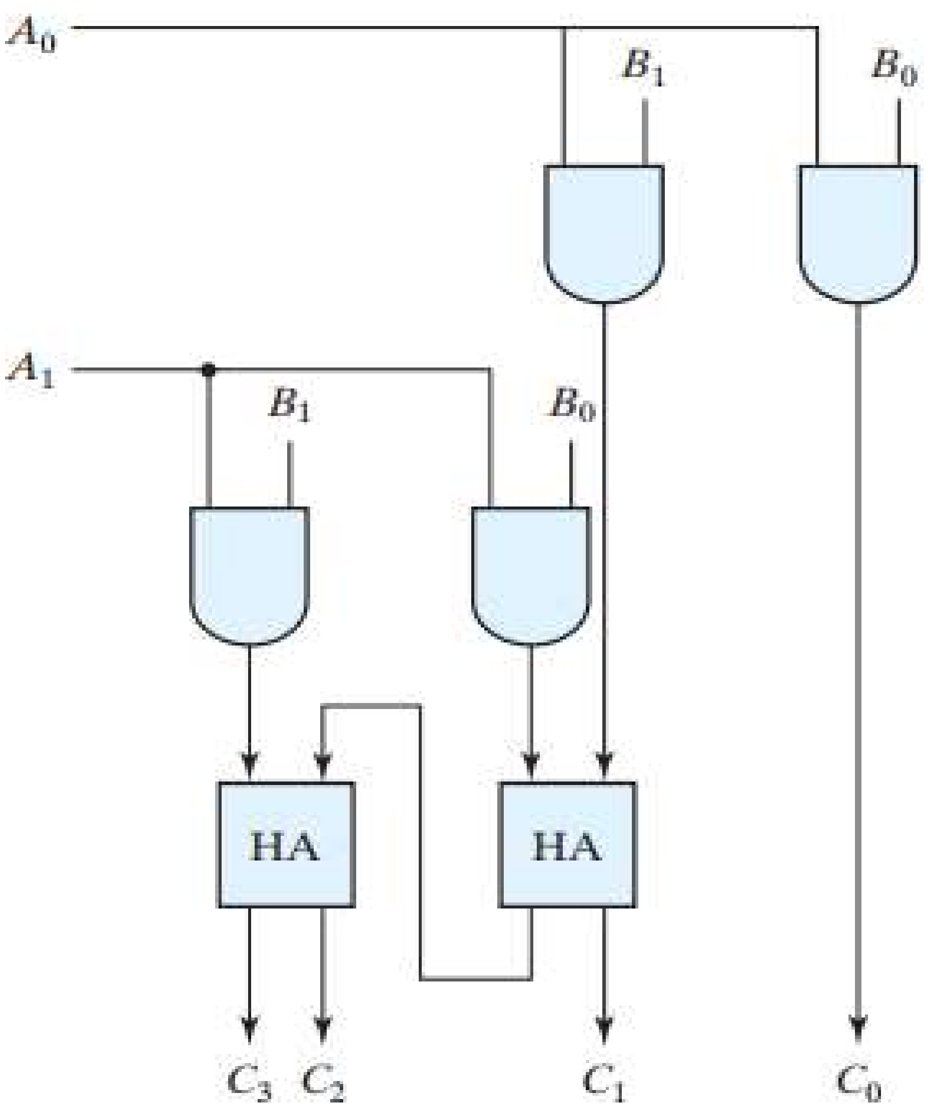
- Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit. Each such multiplication forms a partial product.
- Successive partial products are shifted one position to the left.
- The final product is obtained from the sum of the partial products.

Binary Multiplier - Implementation

Multiplication Process

$$
\begin{array}{cccc}
 & & B_1 & B_0 \\
 & & A_1 & A_0 \\
\cline{3-4}
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
C_3 & C_2 & C_1 & C_0
\end{array}
$$

Binary Multiplier – Block Diagram



Magnitude Comparator

Magnitude Comparator

- The comparison of two numbers is an operation that determines whether one number is greater than, less than, or equal to the other number.
- A *magnitude comparator* is a combinational circuit that compares two numbers A and B and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether $A > B$, $A = B$, or $A < B$.

Four-bit Magnitude Comparator

Consider two numbers, *A* and *B*, with four digits each. Write the coefficients of the numbers in descending order of significance:

$$A = A_3 A_2 A_1 A_0$$
$$B = B_3 B_2 B_1 B_0$$

Each subscripted letter represents one of the digits in the number. The two numbers are equal if all pairs of significant digits are equal: $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$, and $A_0 = B_0$.

Four-bit Magnitude Comparator

When the numbers are binary, the digits are either 1 or 0, and the equality of each pair of bits can be expressed logically with an exclusive-NOR function as

$$x_i = A_iB_i + A_i'B_i' \quad \text{for } i = 0, 1, 2, 3$$

where x_i = 1 only if the pair of bits in position i are equal

- For equality to exist, all x_i variables must be equal to 1, a condition that dictates an AND operation of all variables:

$$(A = B) = x_3x_2x_1x_0$$

Four-bit Magnitude Comparator

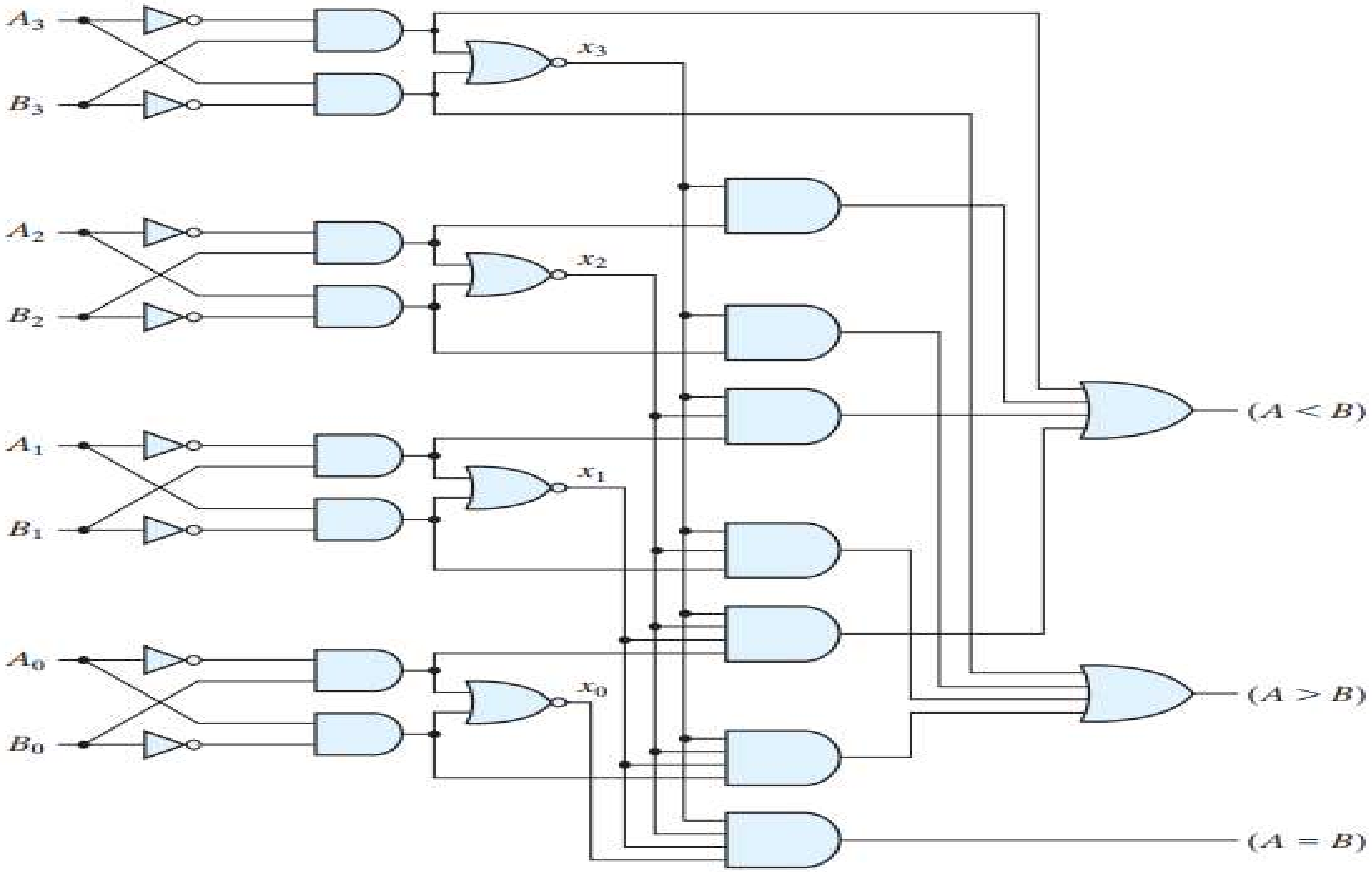
- To determine whether *A* is greater or less than *B*, we inspect the relative magnitudes of pairs of significant digits, starting from the most significant position.
- If the two digits of a pair are equal, we compare the next lower significant pair of digits. The comparison continues until a pair of unequal digits is reached.
- If the corresponding digit of *A* is 1 and that of *B* is 0, we conclude that $A > B$ and vice versa.

Magnitude Comparator

The sequential comparison can be expressed logically by the two Boolean functions

$$(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$
$$(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1' + x_3x_2x_1A'n_0B_0'$$

Magnitude Comparator – Logic Diagram



Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 10

Combinational Logic

Weekly Learning Outcomes

1. Learn the basics of decoders what are they used for, how do they work, and how to create a decoder
2. Learn the basics of encoders what are they used for, how do they work, and how to create an encoder
3. Learn the basics of multiplexers what are they used for, how do they work, and how to create a multiplexer
4. Learn how to design a Boolean function using decoders and multiplexers

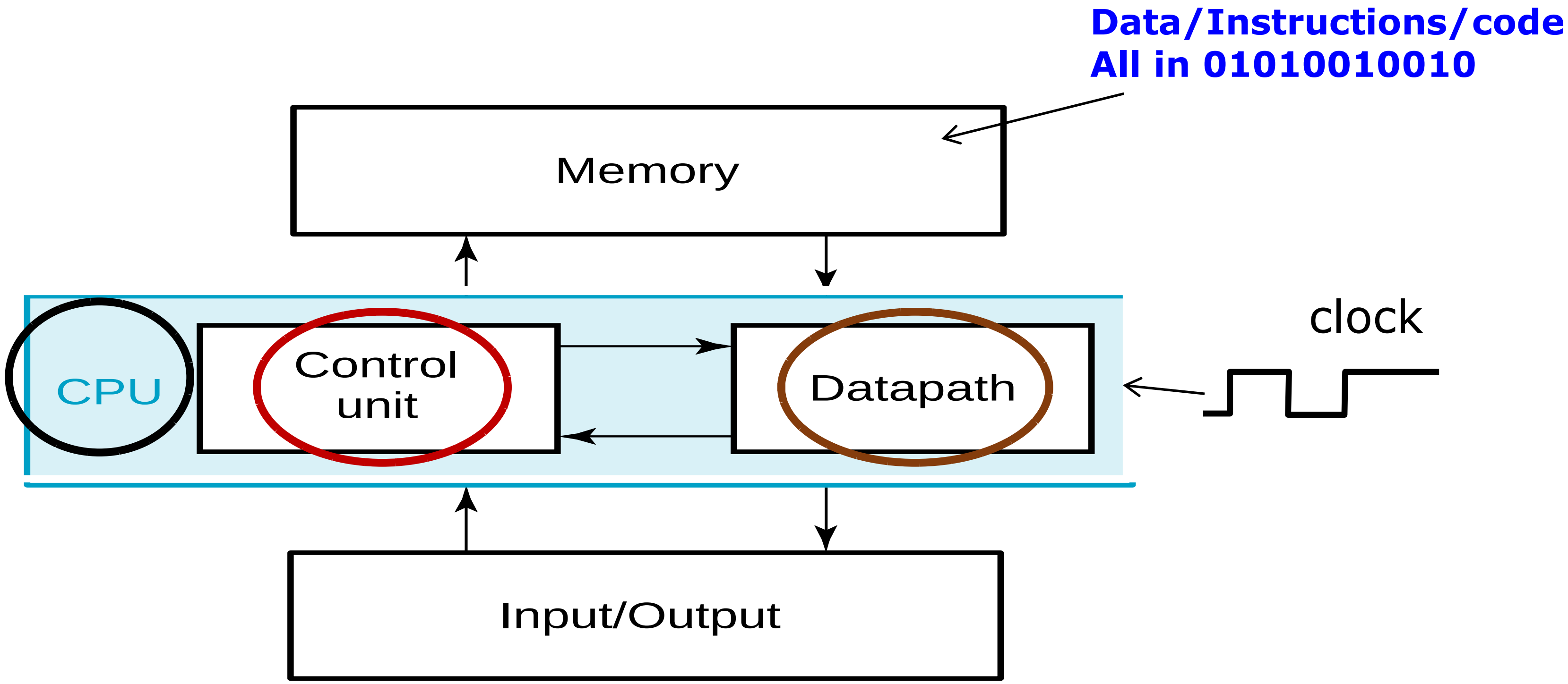
Required Reading

1. Chapter 4 (Sections 4.9- 4.11) (Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapters 12 , Chapter 13 (pp 185-192) (John Seiffertt Digital Logic for Computing)

Central Processing Unit (CPU)



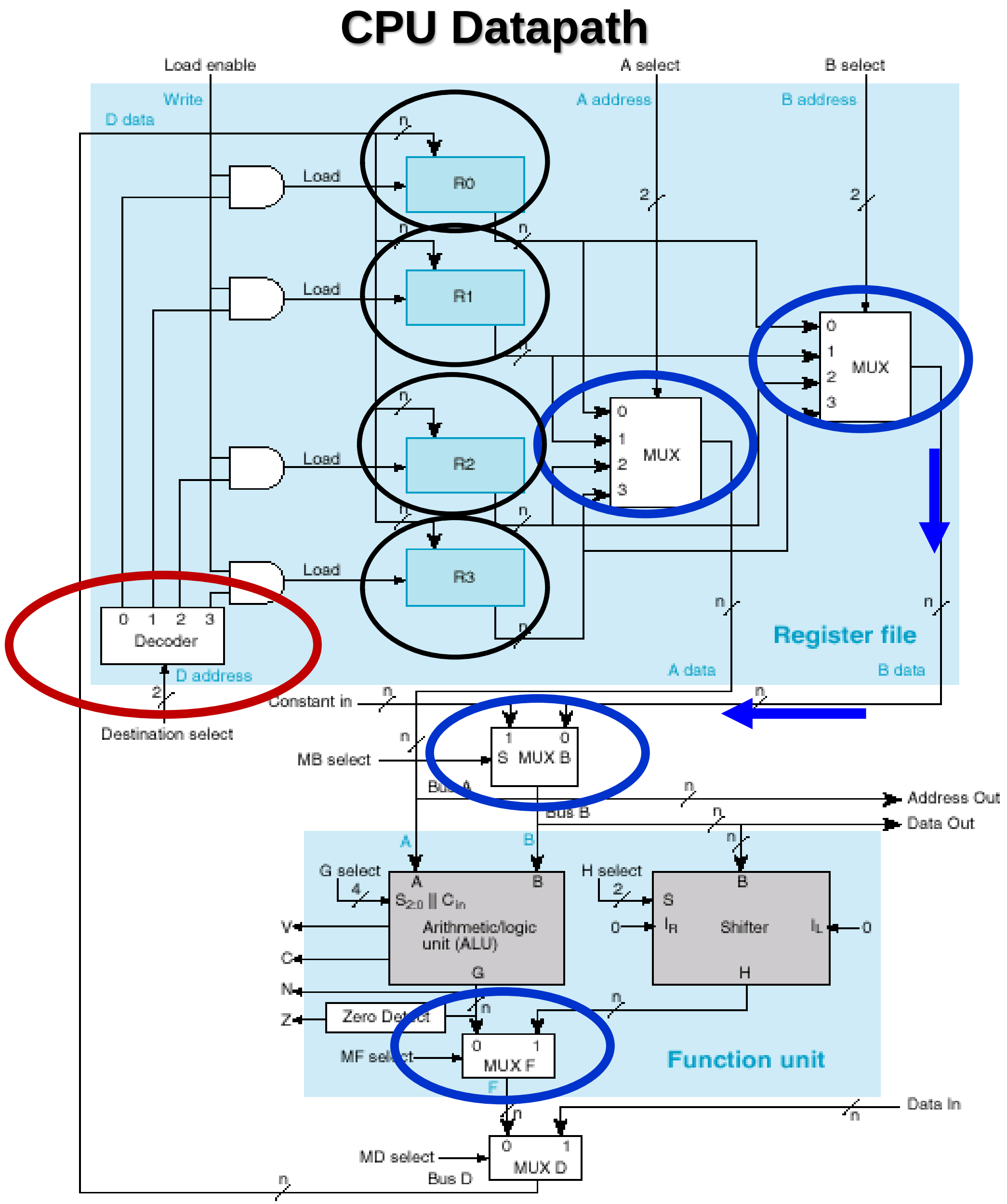
Any CPU consists of two main parts:

- **Datapath Unit**: Arithmetic/Logic Operation
- **Control Unit**: Sends control signals to datapth to perform certain operations.

Source: Dr. S. Areibi class notes

Decoders and Multiplexors

- Decoders and Multiplexors are important components and fundamental circuits that are found in any Central Processing Unit (CPU)
    - Multiplexors are used to select certain registers to act as operands (inputs) to the Arithmetic/Logic Unit
    - Decoders are used to enable registers to write the results back into the registers.

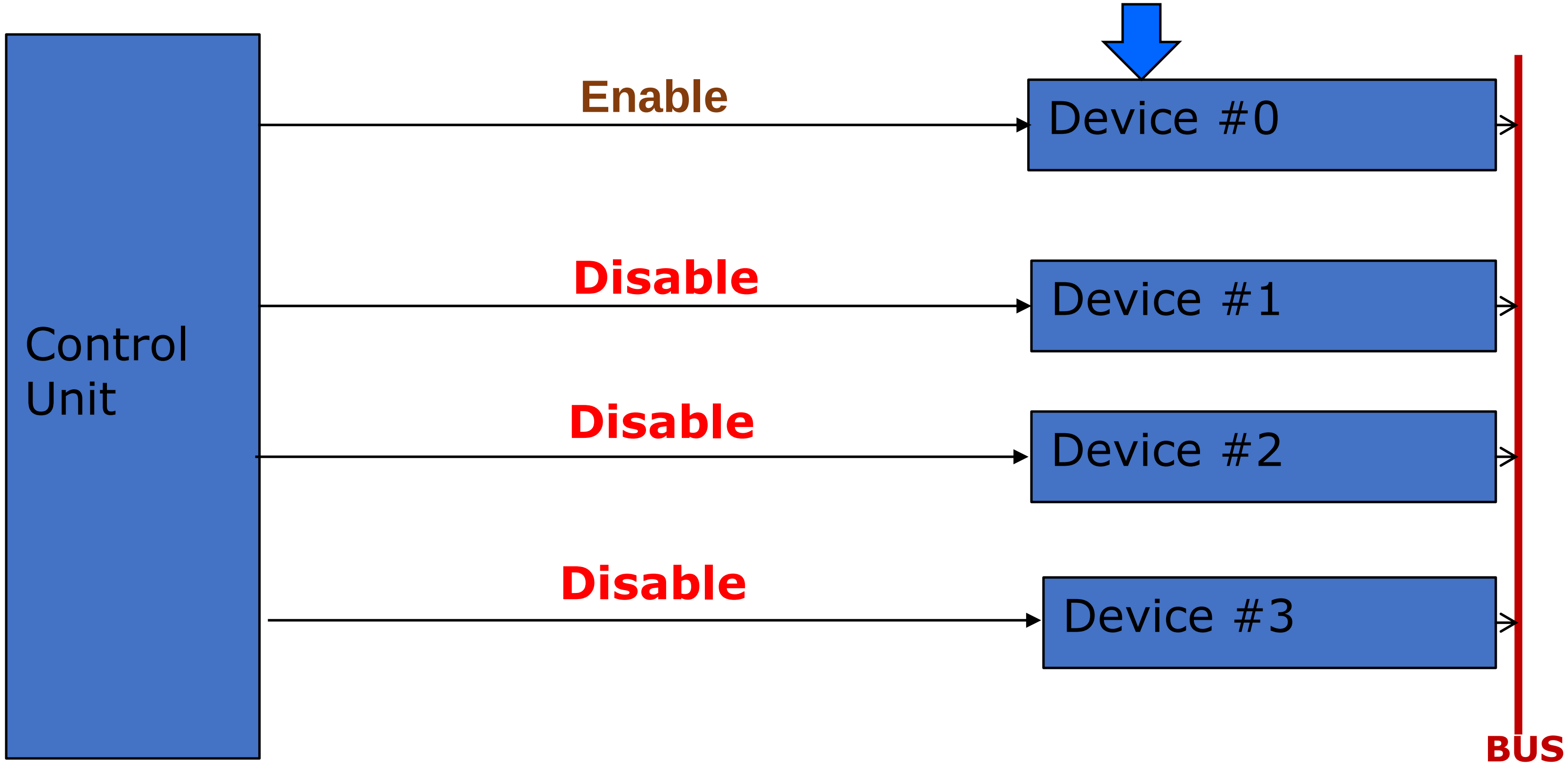


Source: Dr. S. Areibi class notes

- Decoders

Devices on a Bus

- Assume we have 4 devices that needs to put its data on a Bus
- Only one device has to be active at a time (i.e. writes info on the bus)!
- 4 control signals "wires" from the Control Unit are needed to enable each device
- If more devices are to be attached to the bus then we will need many more wires coming out of the control unit!!



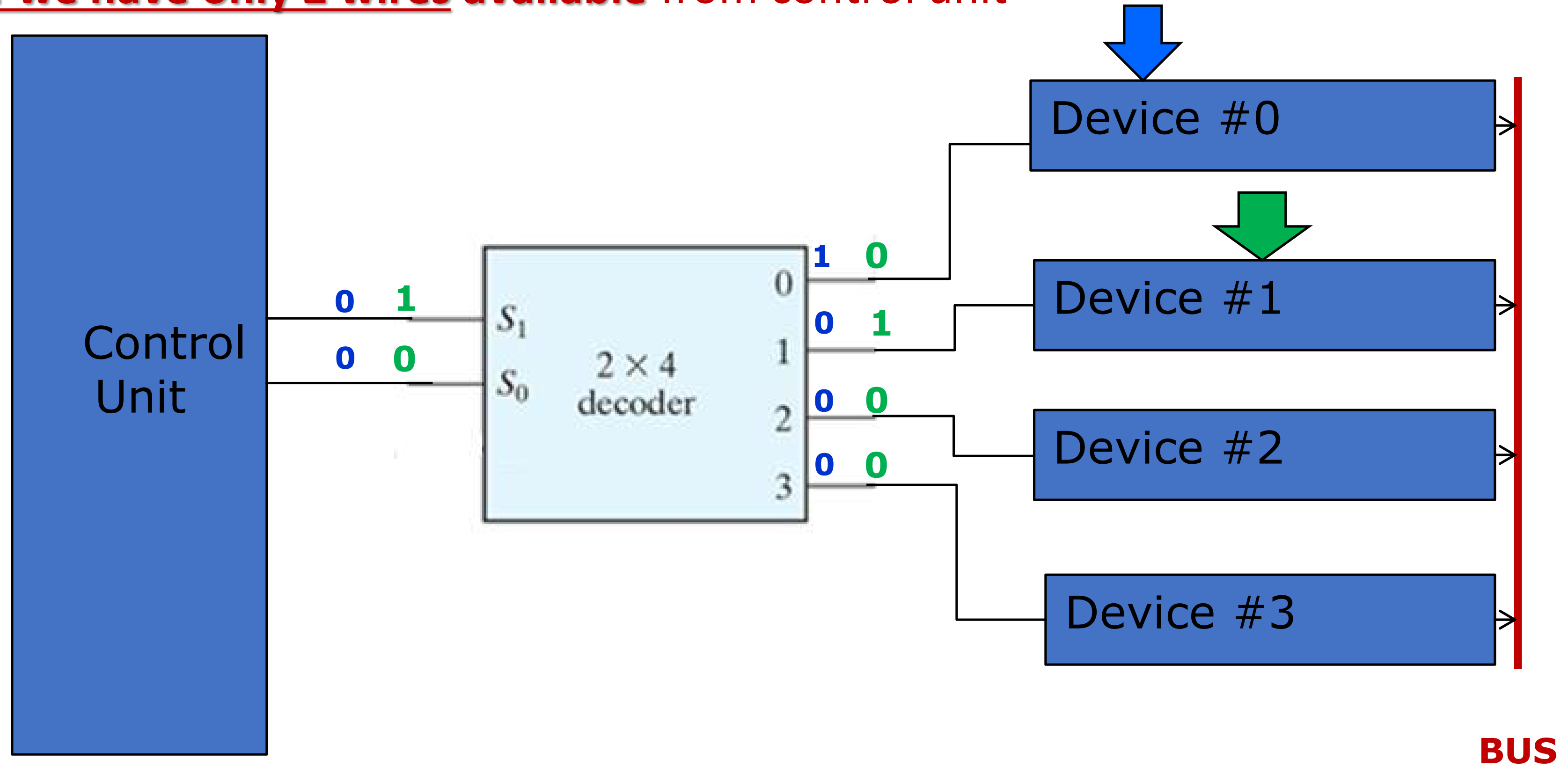
Address Decoding

- 4 devices need to write data to the Bus
- Only one device has to be active at a time!
- But! **What if we have only 2 wires available** from control unit

Truth Table

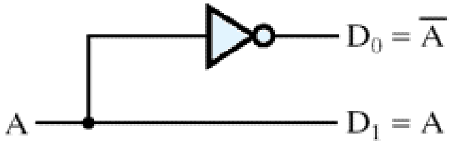
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)

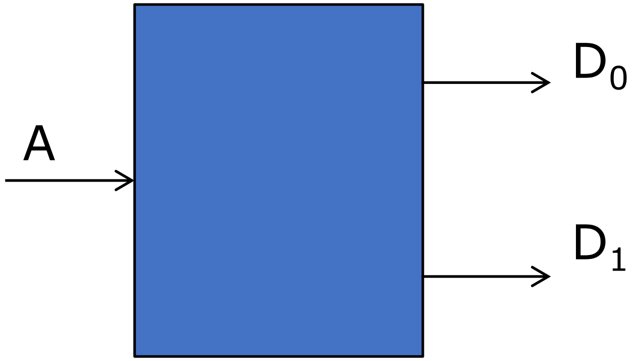


Decoders

- Are circuits with ***n* inputs** and **2^n outputs**
- **Drives high** the **output** corresponding to **binary code of input**
- Several Applications: Address Decoding, …



A	D_0	D_1
0	1	0
1	0	1

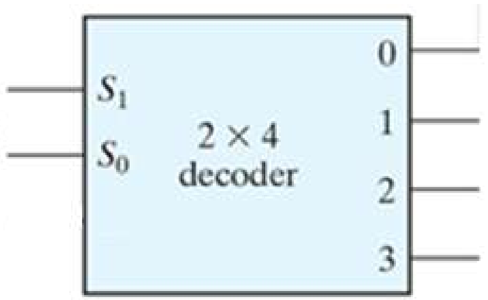


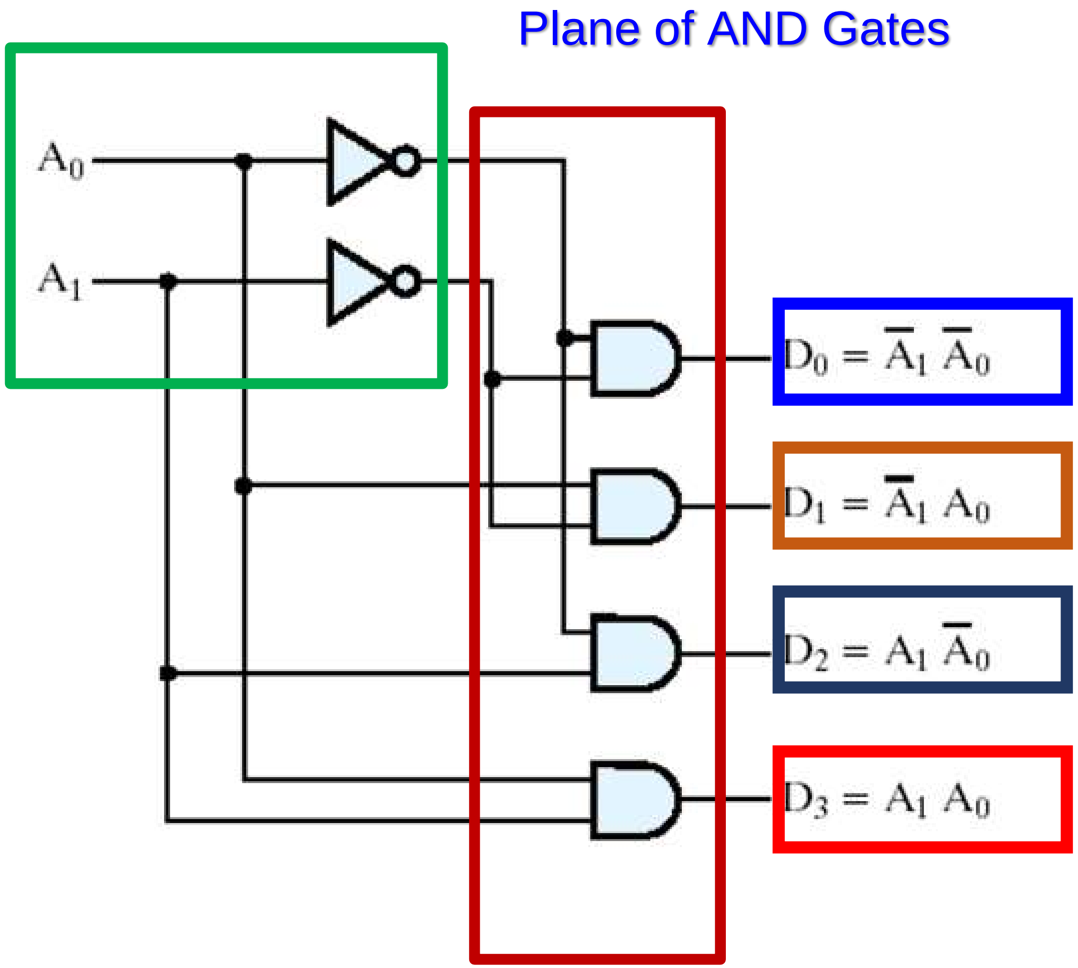
2-to-4 Line Decoder

- Notice they are minterms

Inputs		Outputs			
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)



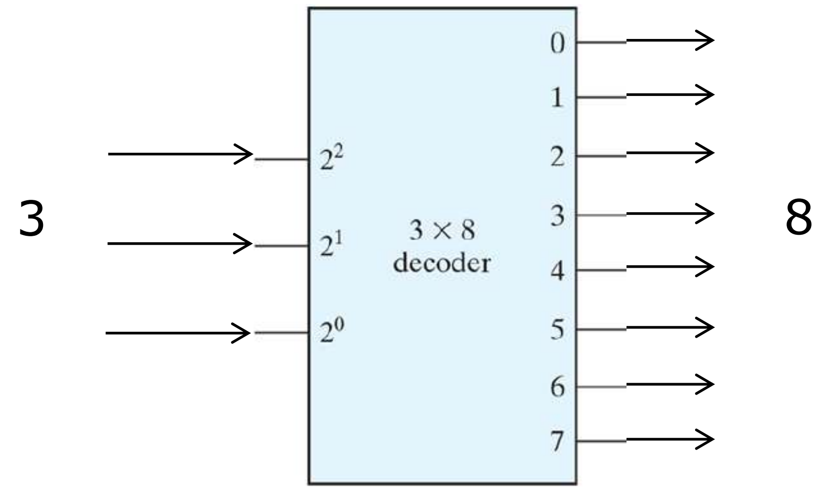


Other Decoders

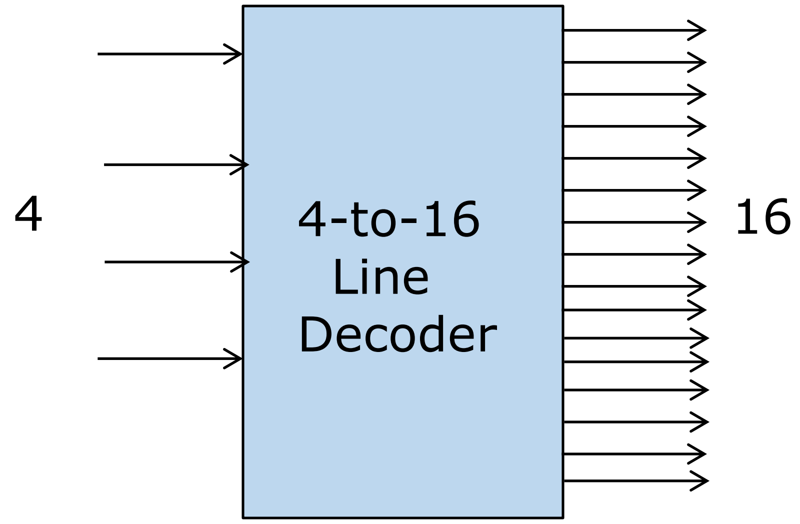
Examples:

- 3-to-8 Decoder
- 4-to-16 Decoder

Binary to Octal,



Binary to Hex, e.t.c

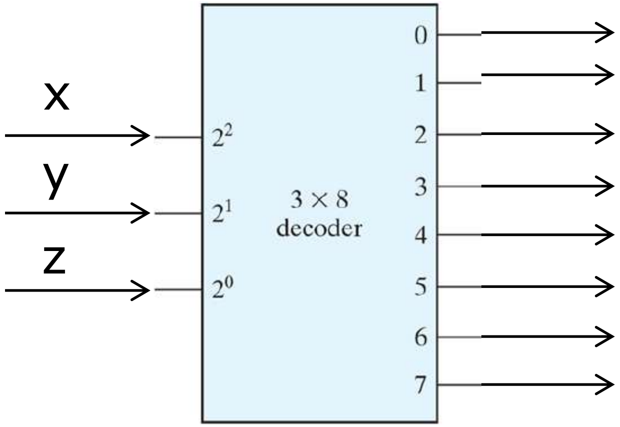


Truth Table, 3-to-8 Decoder

- Example: **Binary to Octal**, Binary to Hex, e.t.c

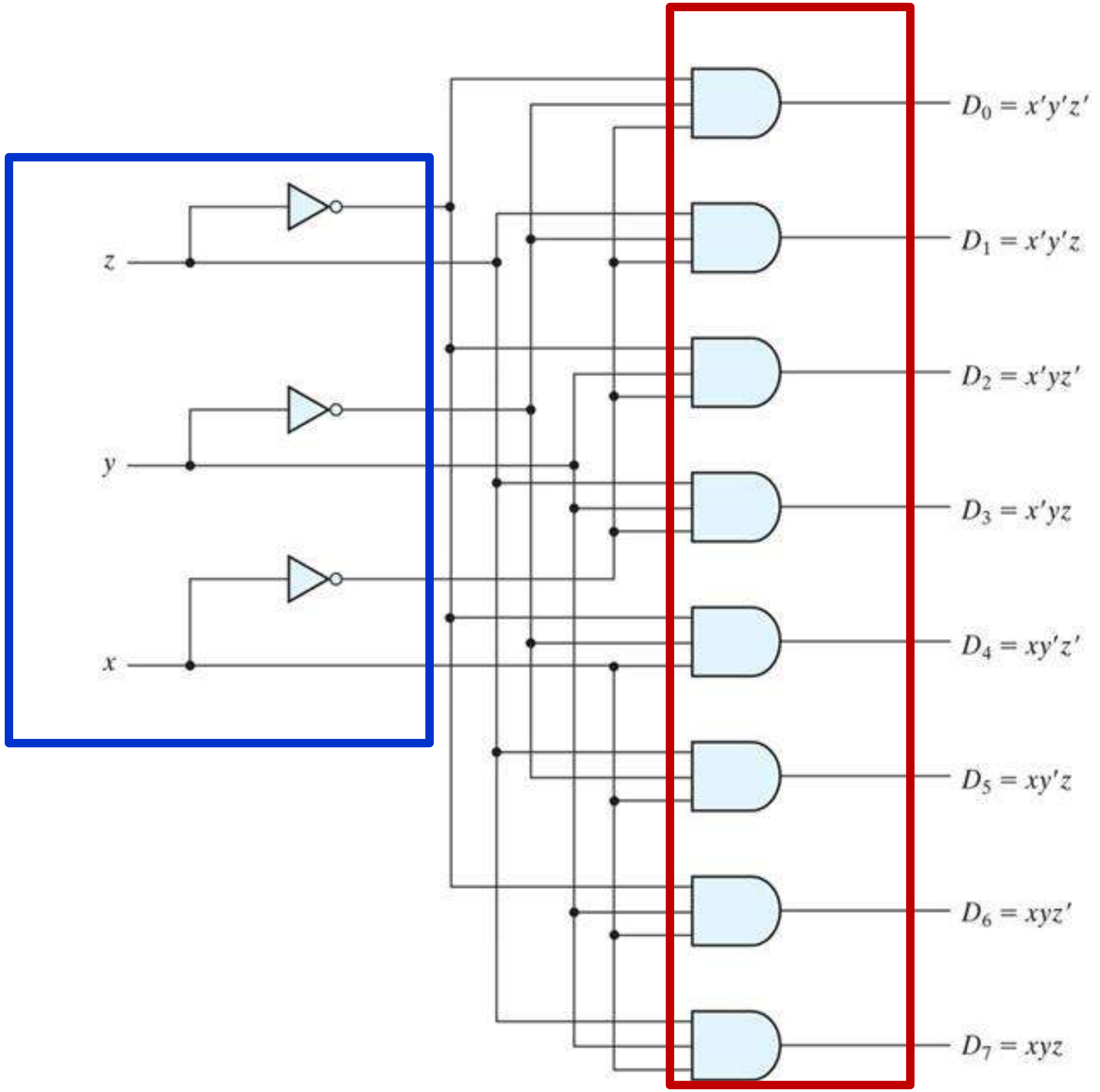
Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Binary to Octal,



Notice they are **minterms**

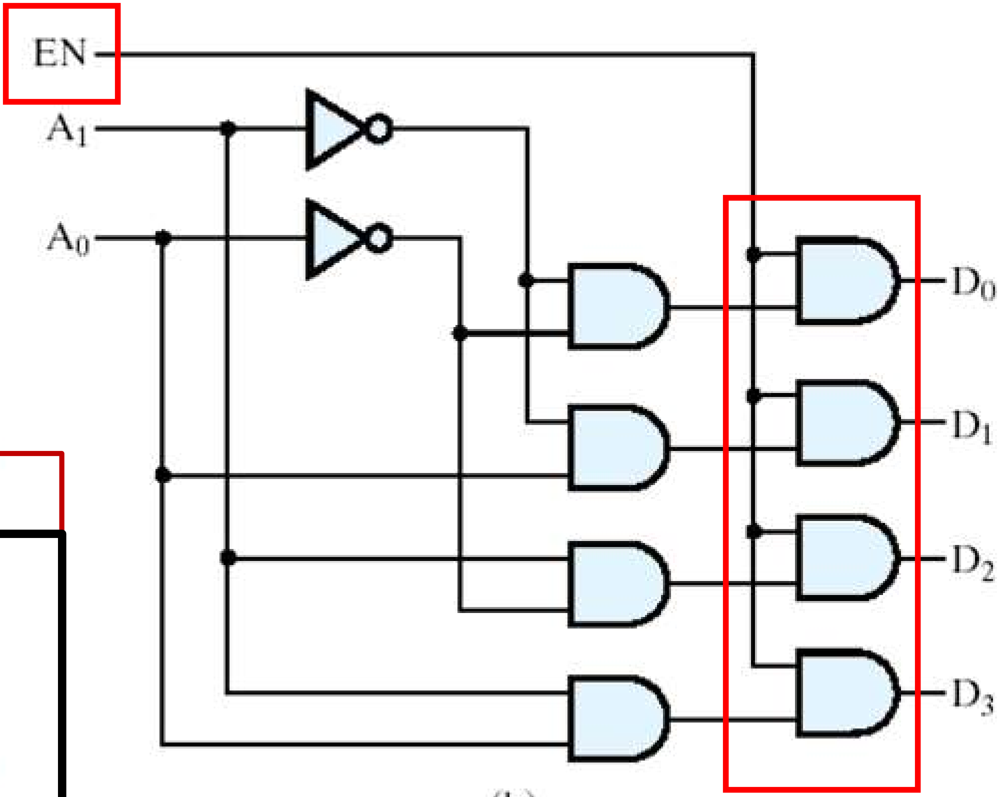
3-to-8 Line Decoder Schematic



2-to-4 with Enable

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(a)



(b)

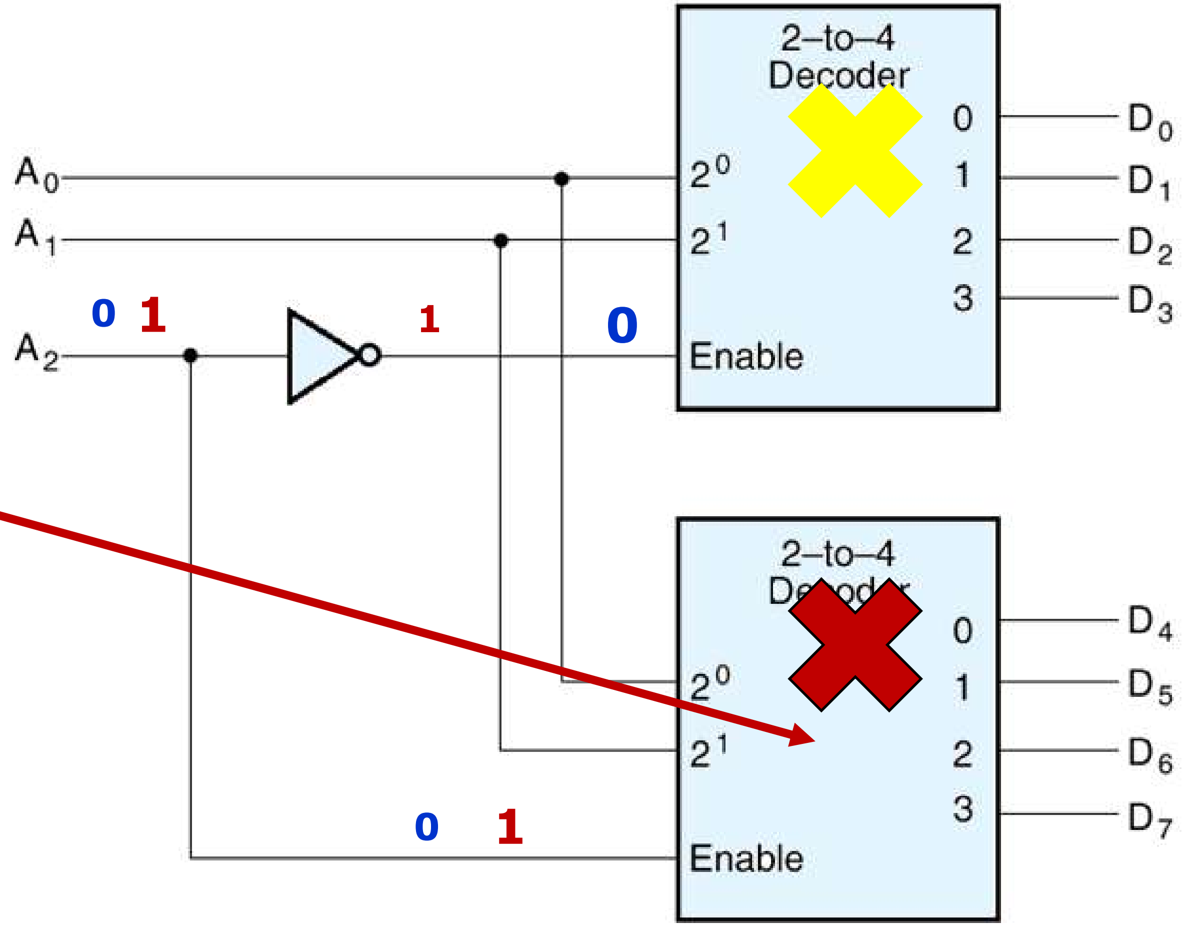
Why use an Enable?

Enable Used for Expansion

A2	A1	A0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

A 3-to-8 Decoder can be realized <u>using two</u> 2-to-4 line decoders with an enable

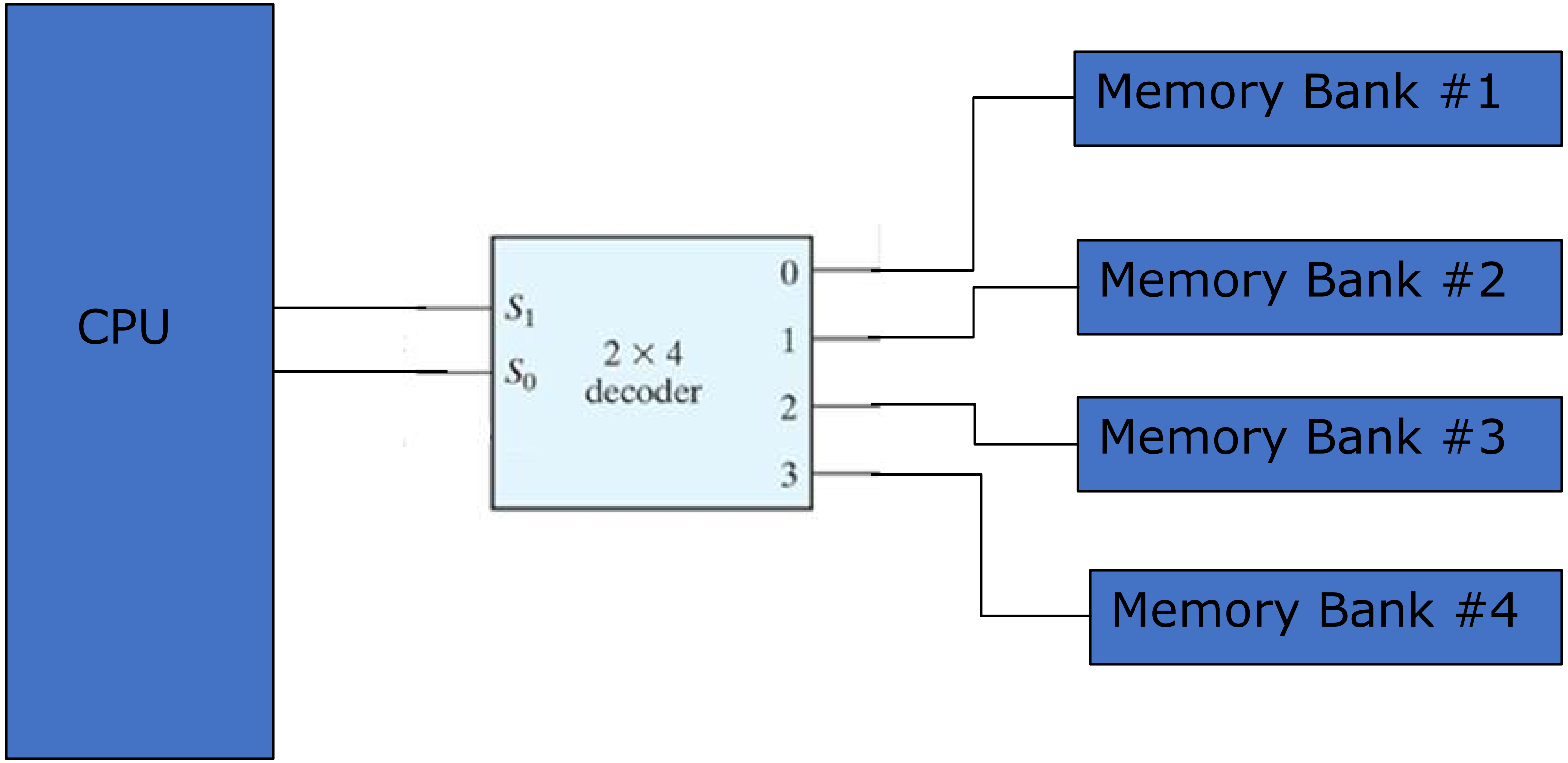
EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



Usage for Decoders

- Binary to Octal/Hex converters.
    - Selecting memory banks, for example 4 memory banks can be selected individually using 2 address lines.
- Implementing logic circuits!
- Decoders are used in Micro Computer Interfacing for Keyboard and Display applications.

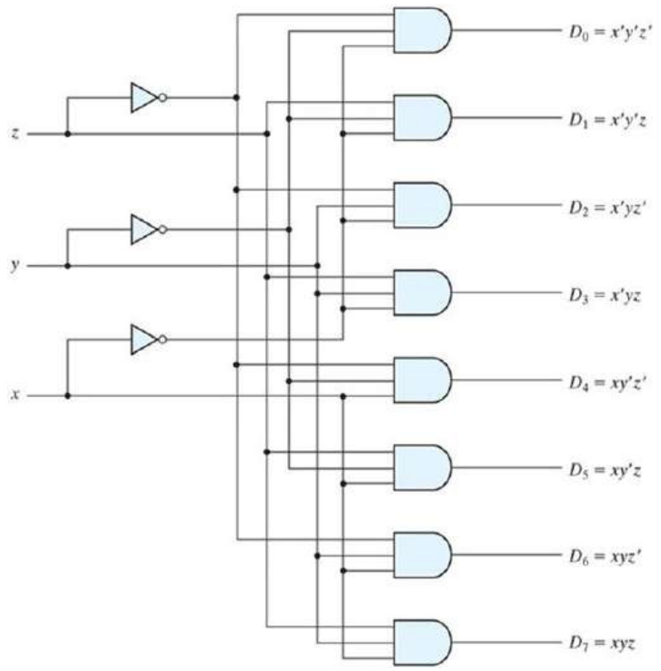
Address Decoding



- Implementing logic with Decoders

Decoders as General-purpose Logic

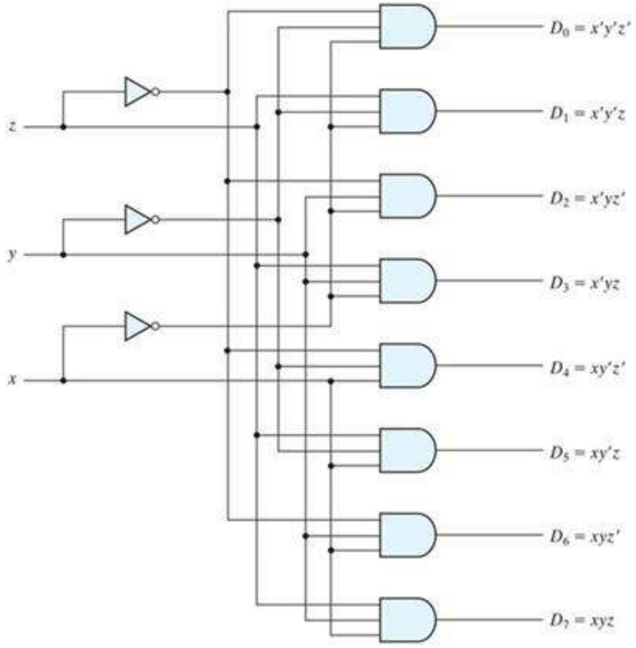
- $n:2^n$ decoder implements any function of n variables
  - With the variables used as control inputs
  - Enable inputs tied to 1 and
  - Appropriate minterms summed to form the function



Decoder generates appropriate minterm based on control signals (it "decodes" control signals)

Decoders as General-purpose Logic

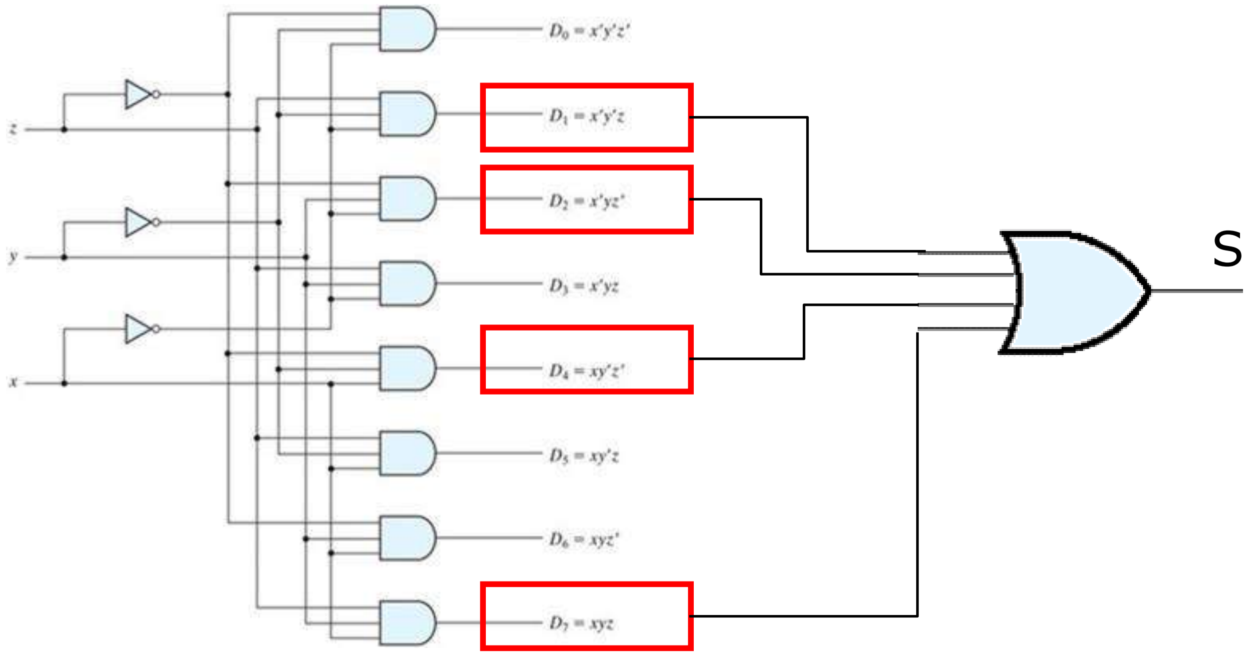
- Example: Implement the following Boolean functions
  - S(x, y, z) = SUM(m(1,2,4,7))
  - C(x, y, z) = SUM(m(3,5,6,7))



1. Since there are three inputs, we need a 3-to-8-line decoder.
2. The decoder generates the eight minterms for inputs x, y, z
3. We choose only the minterms that we need for a function (S) or (C)
4. An OR GATE forms the logical sum minterms required.

Decoders as General-purpose Logic

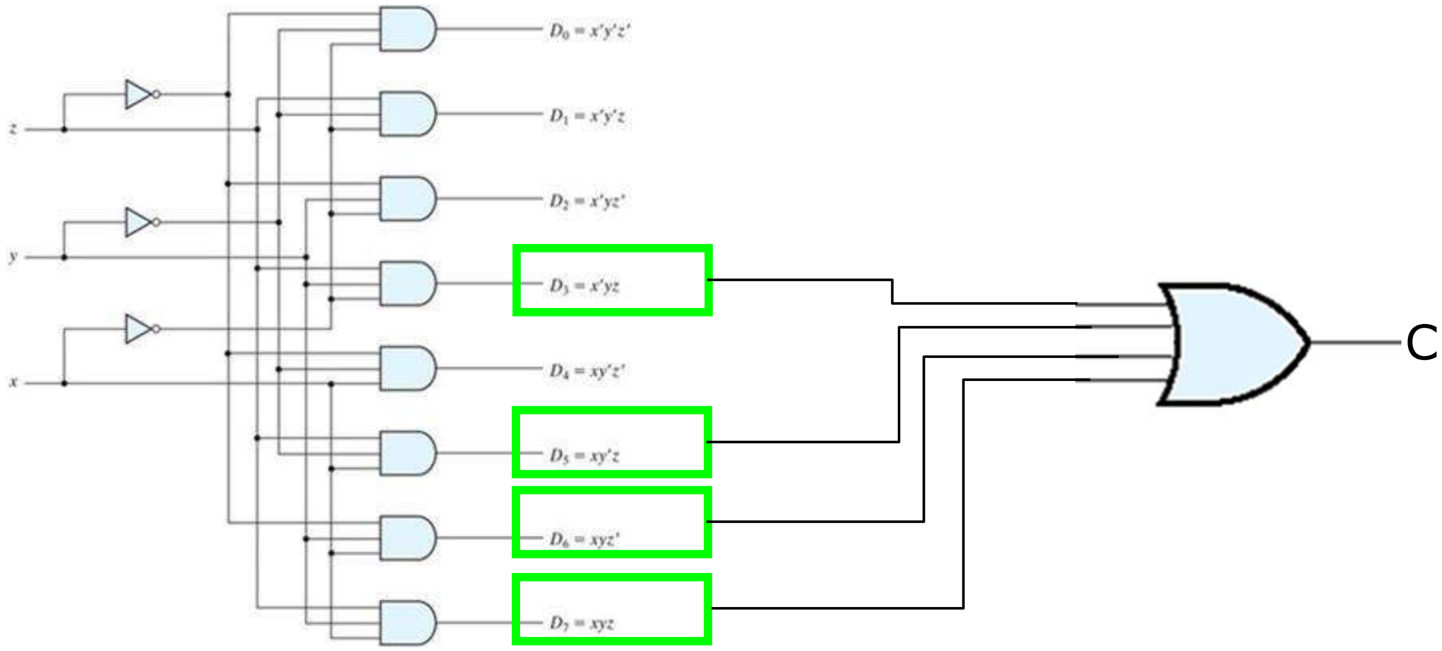
-
  - S(x, y, z) = SUM(m(1,2,4,7))



Decoders as General-purpose Logic

A Single Decoder can implement both “S” and “C” with two OR Gates

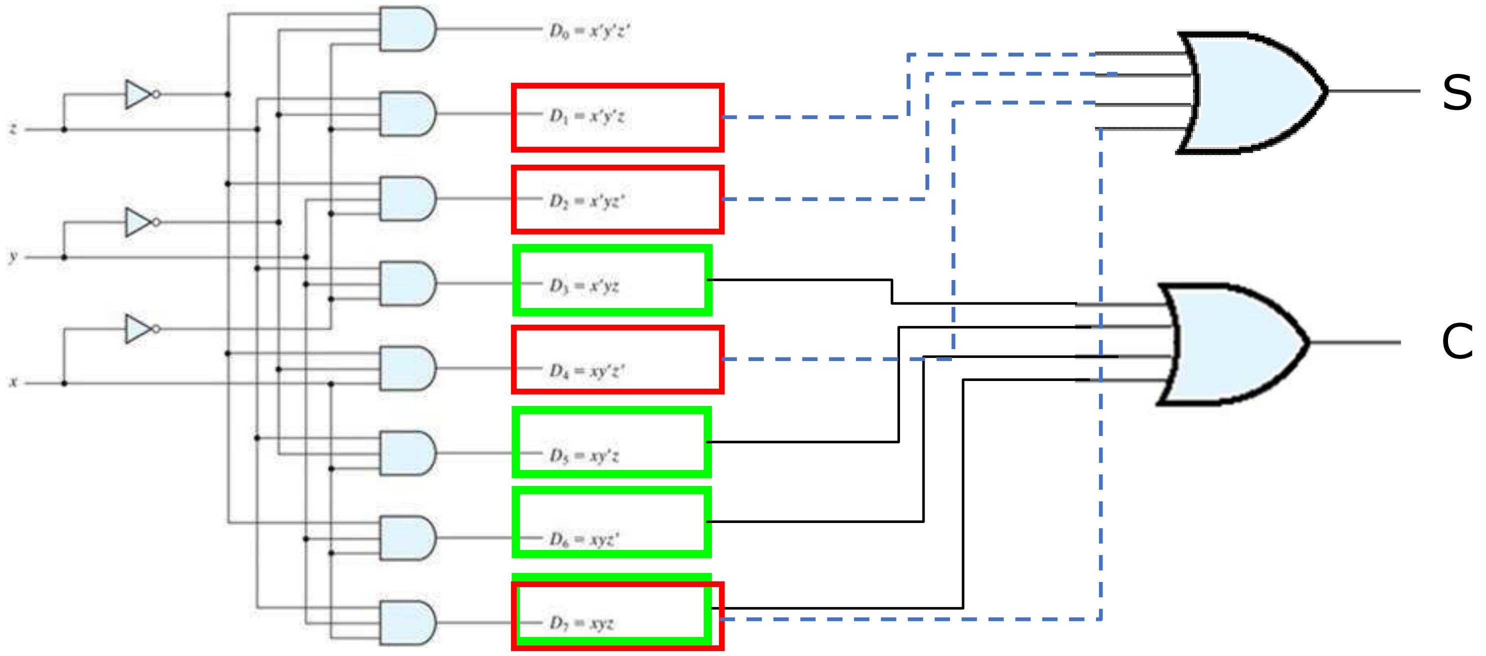
- C(x, y, z) = SUM(m(3,5,6,7))



Decoders as General-purpose Logic

A <u>Single Decoder</u> can implement <u>both</u> “S” and “C” with two OR Gates

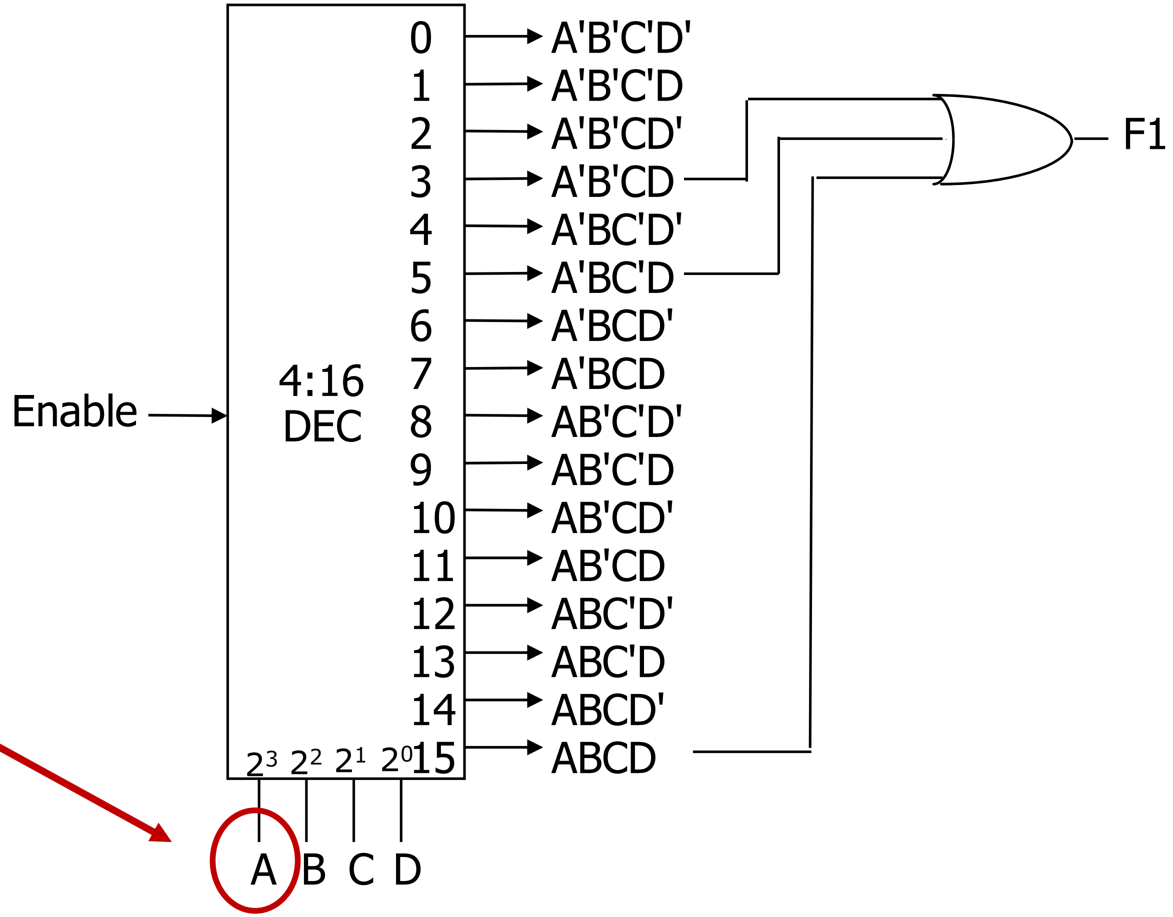
- $S(x, y, z) = SUM(m(1,2,4,7))$
- $C(x, y, z) = SUM(m(3,5,6,7))$



Example

- $F_1 = A'\,B\,C'\,D + A'\,B'\,C\,D + A\,B\,C\,D$
  - $F_1(A,B,C,D) = SUM(m(3,5,15))$

Therefore, we will need a 4-to-16 line decoder



Need to show that variable A (MSB) needs to be connected to MSB of the Decoder

- Encoders

Encoder

- Encoder is the opposite of decoder
  - 2^n inputs (or less – maybe BCD in)
  - n outputs
- Examples:
  - Octal to binary conversion
  - Hexadecimal to binary conversion

Octal to Binary Encoder

You can think of the inputs as the 8 Octal digits

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Only one input is active at a time

Design of Encoder

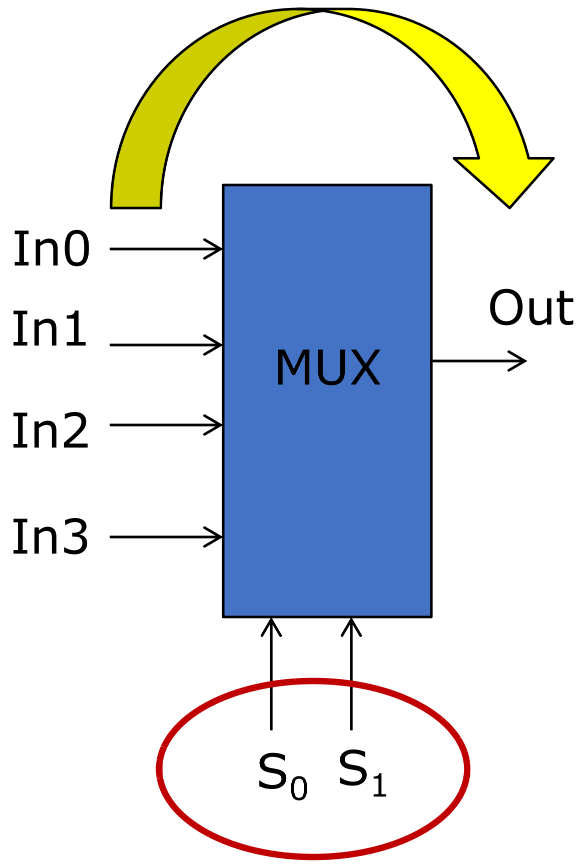
- z = D1 + D3 + D5 + D7

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

- Multiplexors

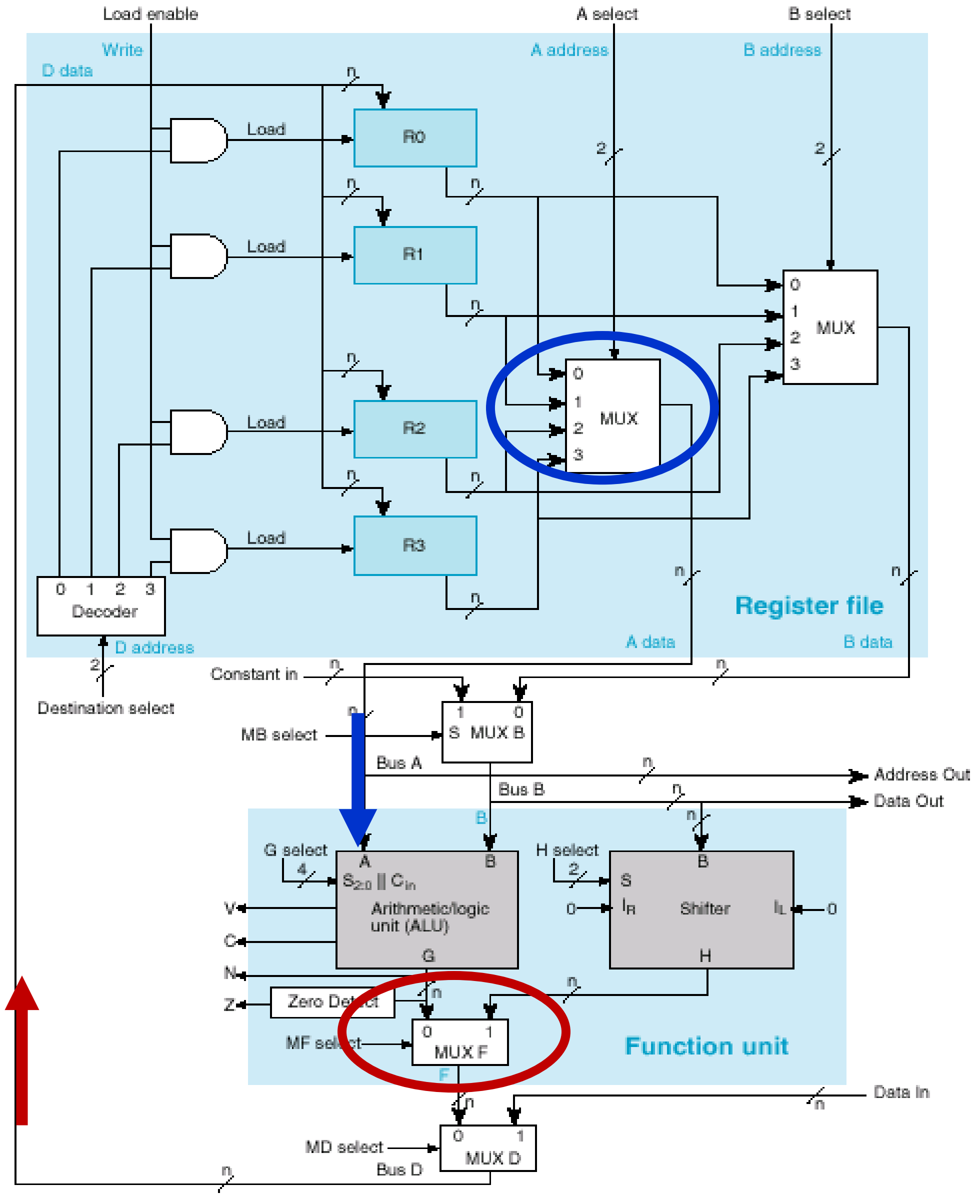
Multiplexer (or Mux)

- **<u>Selects</u>** one of a set of inputs to pass on to output
- For Every 2^n inputs <u>we need</u> n **<u>select lines</u>**
- **<u>Applications</u>**: Useful for choosing from sets of data
    - Memory or register to ALU



Multiplexors

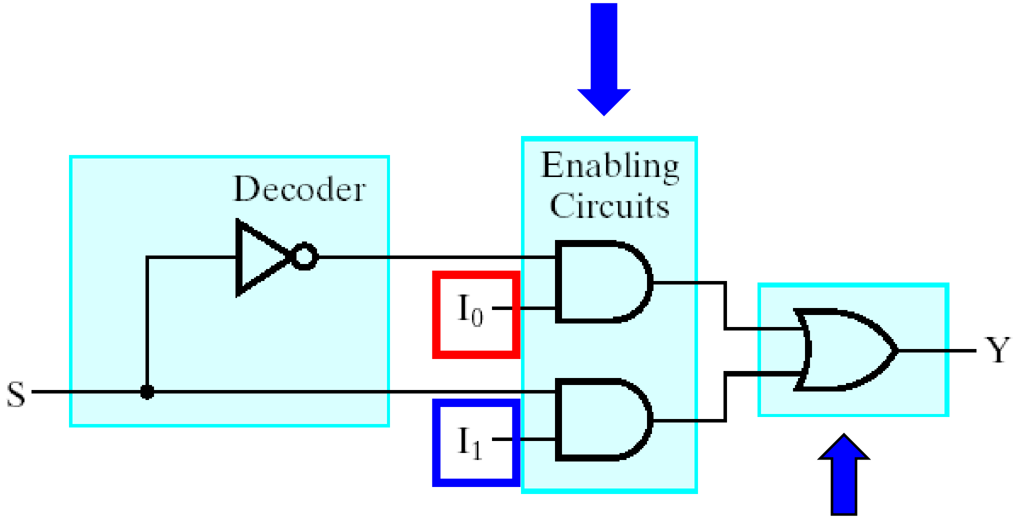
- The Multiplexor in the register file allows one of the four registers R0, R1, R2, R3 to move its contents to the Function Unit
- The Multiplexor in the Function Unit allows the user to choose between the ALU and Shifter to move data back to the Register File.



2-Input Multiplexer

Structure:

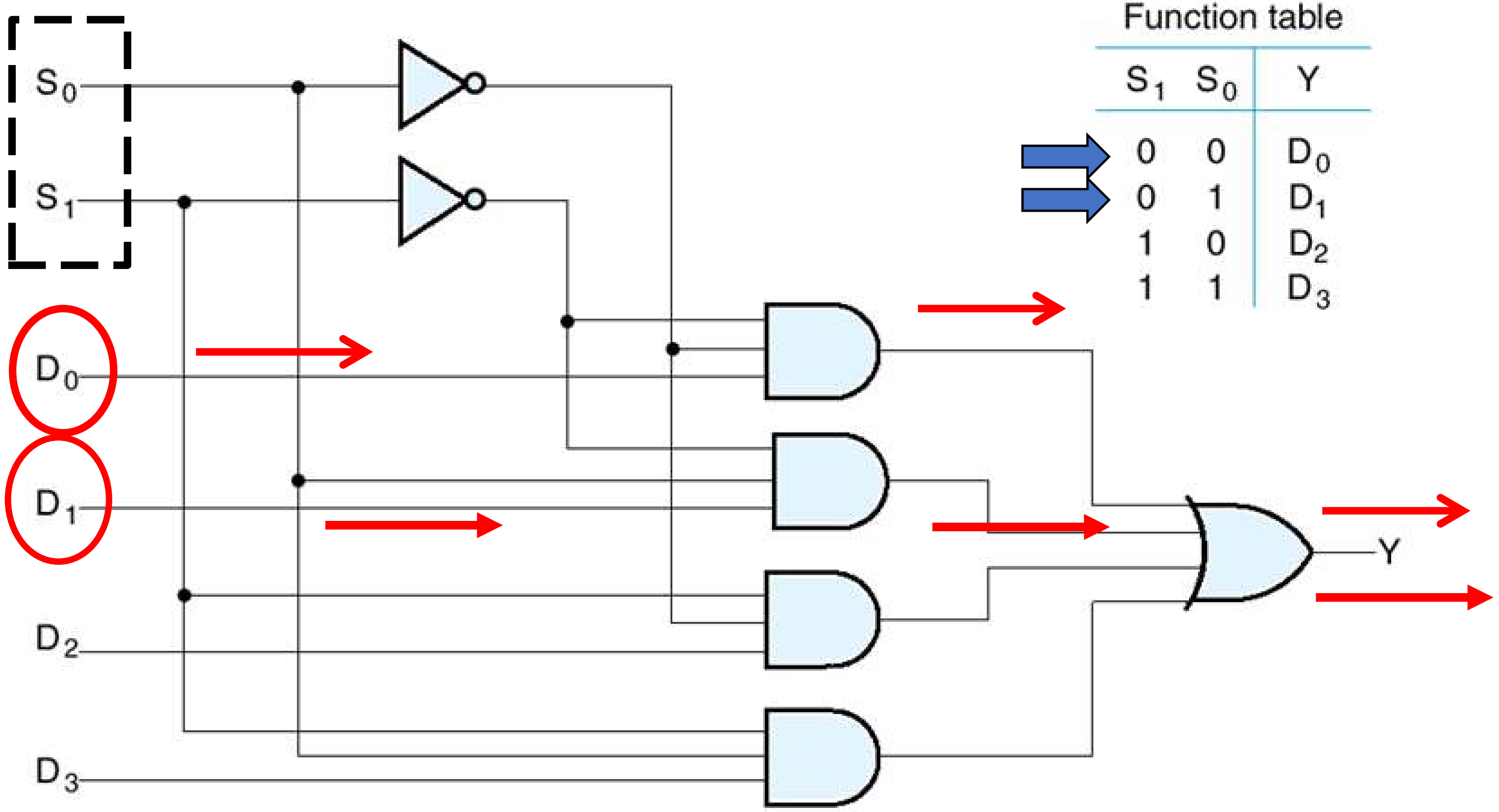
A Plane of AND Gates



Followed by an OR Gate

S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

4-to-1 Line Multiplexer

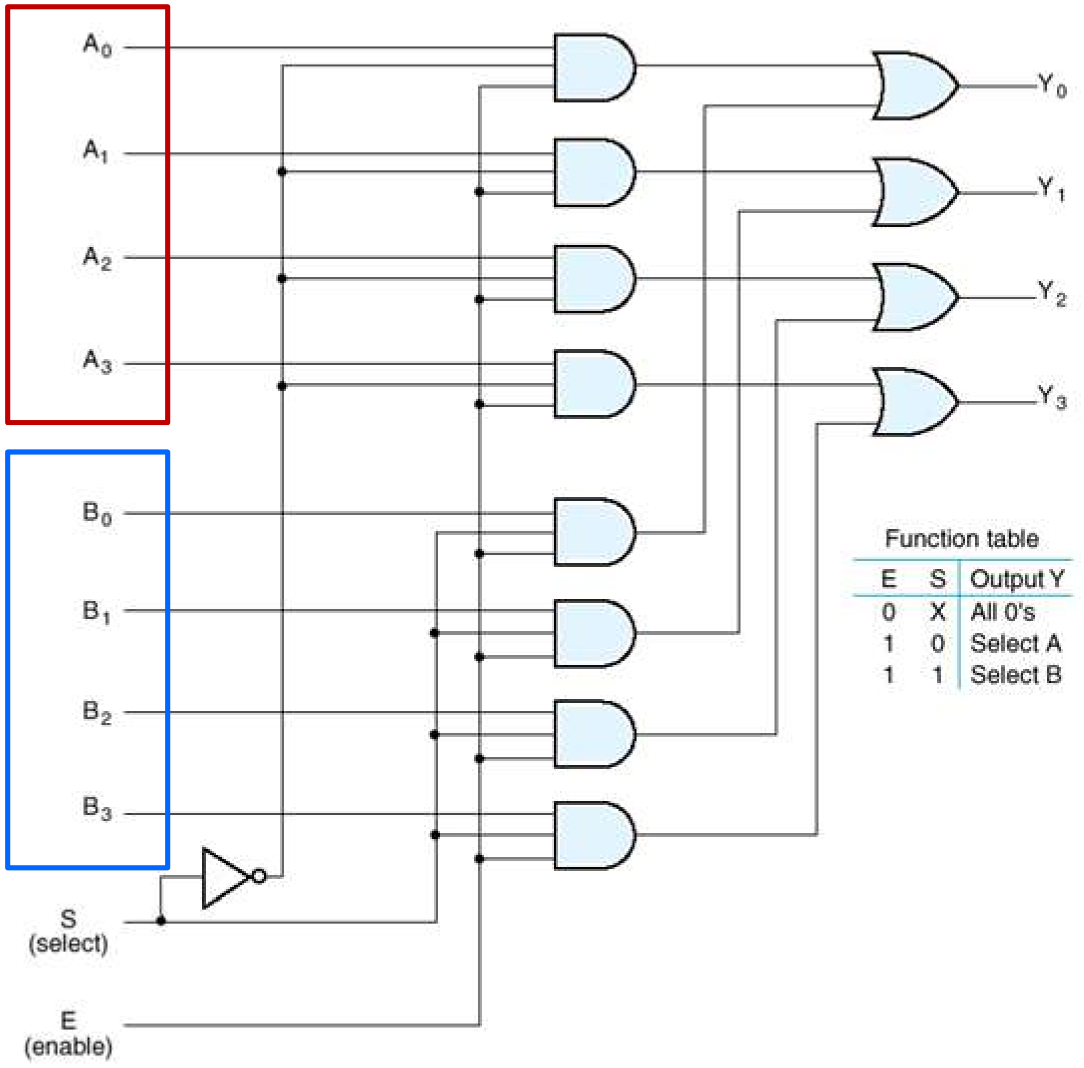


Function table

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Quad 2-to-4 Line Mux

- Select one set of 4 lines



Function table

E	S	Output Y
0	X	All 0's
1	0	Select A
1	1	Select B

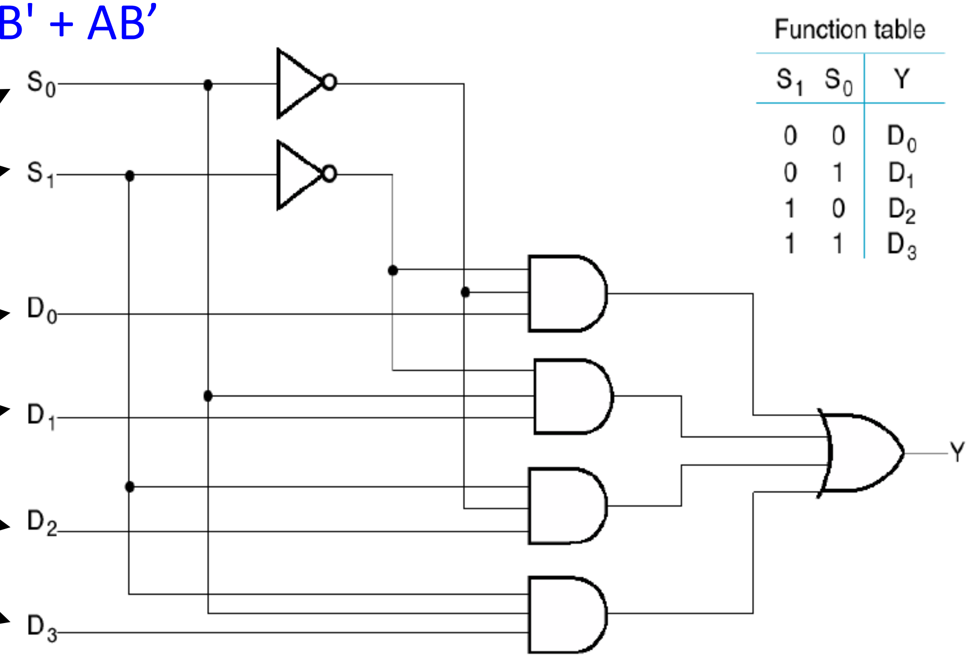
- Implementing logic with Multiplexer

Implementing Logic

- 2^n:1 multiplexer implements any function of n variables
  1. With the variables used as control inputs and
  2. Data inputs tied to 0 or 1 (The D0-D3 are wired to the values of F as logic High or Low)
  3. In essence, *a lookup table*
- Example: F(A,B) = m0 + m2

  = A'B' + AB’

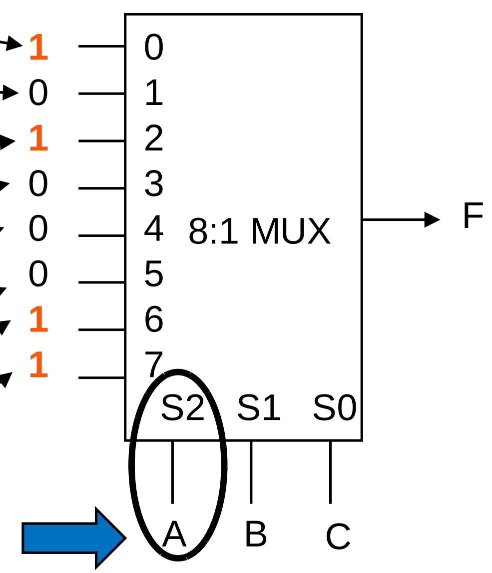
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0



Muxes as General-purpose Logic

- 2^n:1 multiplexer implements any function of n variables
  - With the variables used as control inputs and
  - Data inputs tied to 0 or 1
  - In essence, a lookup table
- Example: F(A,B,C) = m0 + m2 + m6 + m7
  = A'B'C' + A'BC' + ABC' + ABC

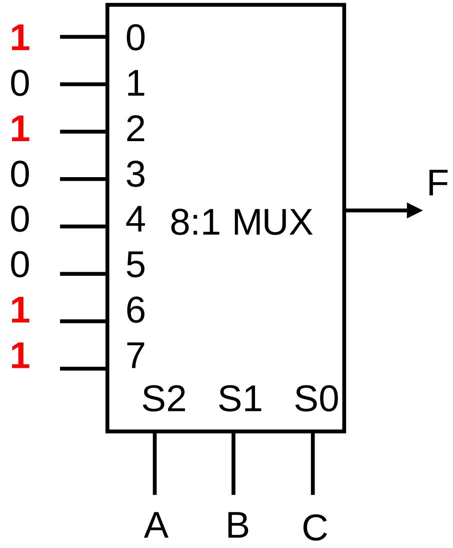
A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Muxes as General-purpose Logic

- 2^{n-1}:1 mux can implement any function of n variables
  - With n-1 variables used as control inputs and
  - Data inputs tied to the last variable or its complement
- Example:
  - F(A,B,C) = m0 + m2 + m6 + m7
    = A'B'C' + A'BC' + ABC' + ABC

Not Optimized



How to use a 4:1 MUX Instead?

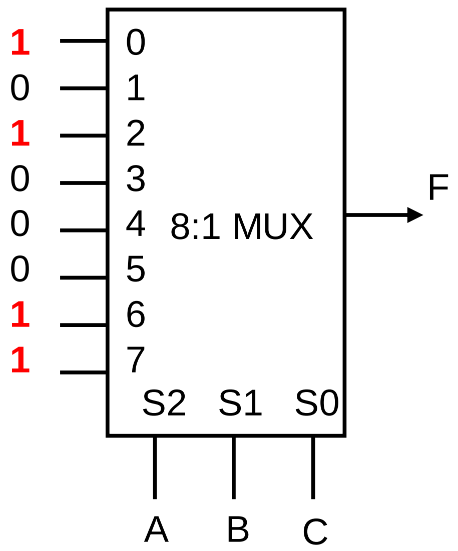
F	C	B	A
1	0	0	0
0	1	0	0
1	0	1	0
0	1	1	0
0	0	0	1
0	1	0	1
1	0	1	1
1	1	1	1

- Use inputs A, B as the selection lines for the 4:1 MUX since we will only have two selection lines
- Find a relationship between C (the third input) and F

Muxes as General-purpose Logic

- 2^{n-1}:1 mux can implement any function of n variables
  - With n-1 variables used as control inputs and
  - Data inputs tied to the last variable or its complement
- Example:
  - F(A,B,C) = m0 + m2 + m6 + m7
    = A'B'C' + A'BC' + ABC' + ABC

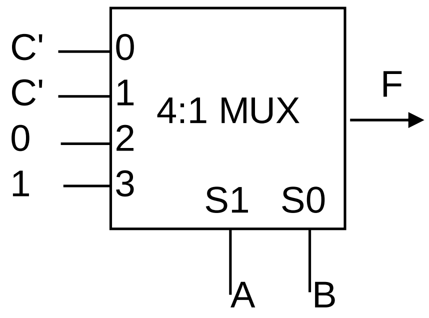
Not Optimized



How to use a 4:1 MUX Instead?

F	C	B	A	
1	0	0	0	C'
0	1	0	0	
1	0	1	0	C'
0	1	1	0	
0	0	0	1	0
0	1	0	1	
1	0	1	1	1
1	1	1	1	

Optimized



Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 11

Synchronous Sequential Logic

CONTENTS

- Sequential Circuits
- Storage Elements: Latches
- Storage Elements: Flip-Flops

Weekly Learning Outcomes

1. Know how to distinguish a sequential circuit from a combinational circuit.
2. Understand the functionality of a SR latch, transparent latch, D flipflop, JK flip-flop, and T flip-flop.
3. Know how to use the characteristic table and characteristic equation of a flip-flop.

Required Reading

1. Chapter 5 (5.1 to 5.4)
(Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapter 11

(Digital Logic for Computing) (John Seiffertt Digital Logic for Computing)

Sequential Circuits

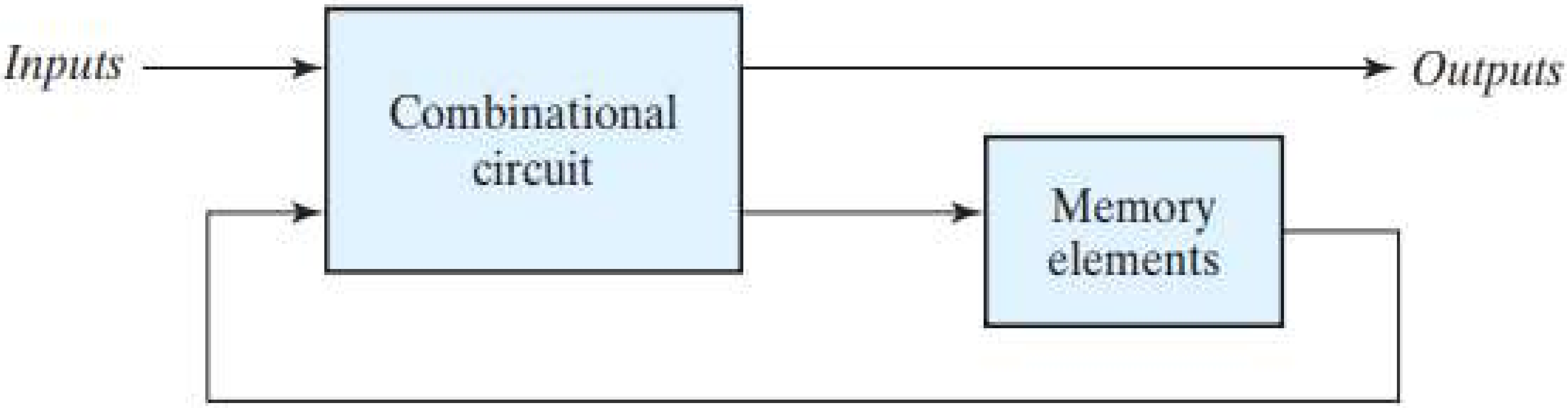
Sequential Circuits - Introduction

- The digital circuits considered so far have been combinational—their output depends only and immediately on their inputs—they have no memory, i.e., dependence on past values of their inputs.
- Sequential circuits, however, act as storage elements and have memory. They can store, retain, and then retrieve information when needed at a later time.

Sequential Circuits - Introduction

- Hand-held devices, cell phones, navigation receivers, personal computers, digital cameras, personal media players, and virtually all electronic consumer products have the ability to send, receive, store, retrieve, and process information represented in a binary format.
- The technology enabling and supporting these devices is critically dependent on electronic components that can store information, i.e., have memory.

Sequential Circuits – Block Diagram



Sequential Circuits

- It consists of a combinational circuit to which storage elements are connected to form a feedback path.
- The storage elements are devices capable of storing binary information.
- The binary information stored in these elements at any given time defines the *state* of the sequential circuit at that time.

Sequential Circuits

- The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements.
- The next state of the storage elements is also a function of external inputs and the present state. Thus, **a sequential circuit is specified by a time sequence of inputs, outputs, and internal states**.

Sequential Circuits - Types

- A ***synchronous* sequential** circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time.

- The behavior of an ***asynchronous* sequential** circuit depends upon the input signals at any instant of time *and* the order in which the inputs change.

Synchronous Sequential Circuits

- A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time.
- Synchronization is achieved by a timing device called a *clock generator*, which provides a clock signal having the form of a periodic train of *clock pulses*.
- The clock signal is commonly denoted by the identifiers *clock* and *clk*.

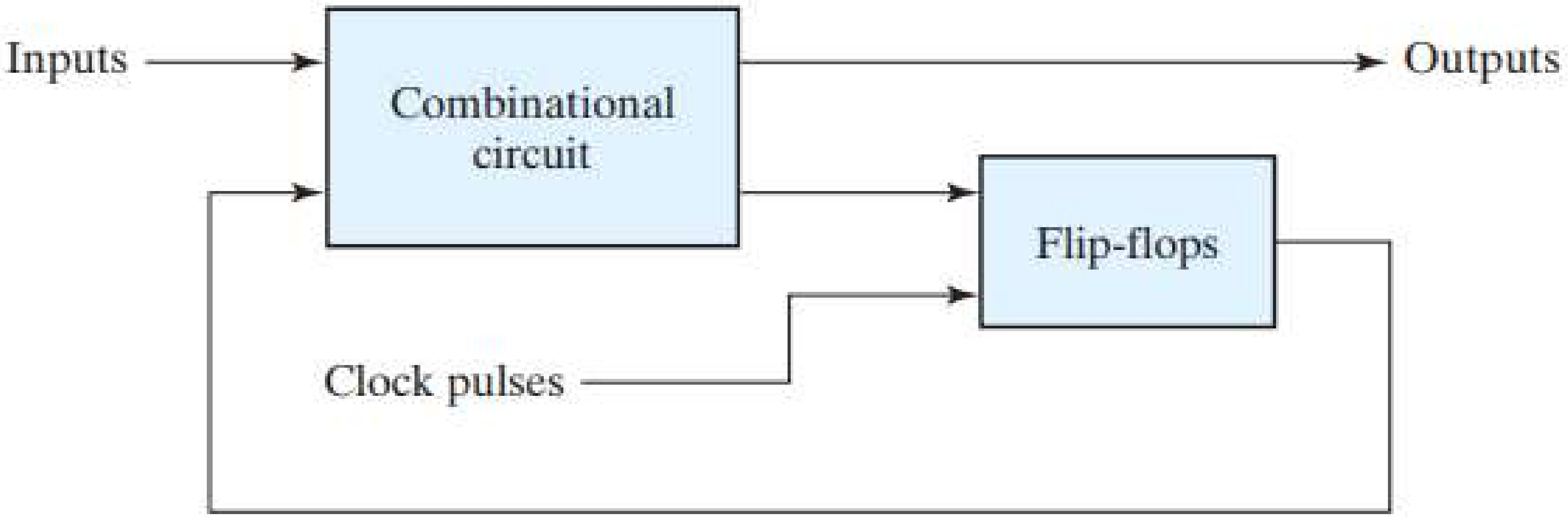
Sequential Circuits

- The clock pulses determine ***when*** computational activity will occur within the circuit, and other signals (external inputs and otherwise) determine ***what*** changes will take place affecting the storage elements and the outputs.
- For example, a circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the occurrence of a clock pulse. Synchronous sequential circuits that use clock pulses to control storage elements are called ***clocked sequential circuits***

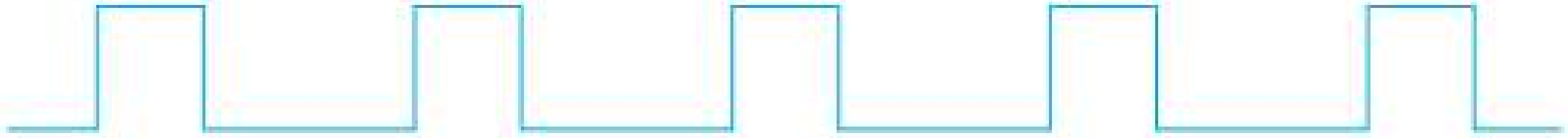
Flip Flops

- The storage elements (memory) used in clocked sequential circuits are called *flipflops.*
- A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1.
- A sequential circuit may use many flip-flops to store as many bits as necessary.
- The block diagram of a **synchronous clocked sequential circuit** is shown in next slide.

Flip Flops



(a) Block diagram



(b) Timing diagram of clock pulses

Sequential Circuits: Latches

Storage Elements: Latches

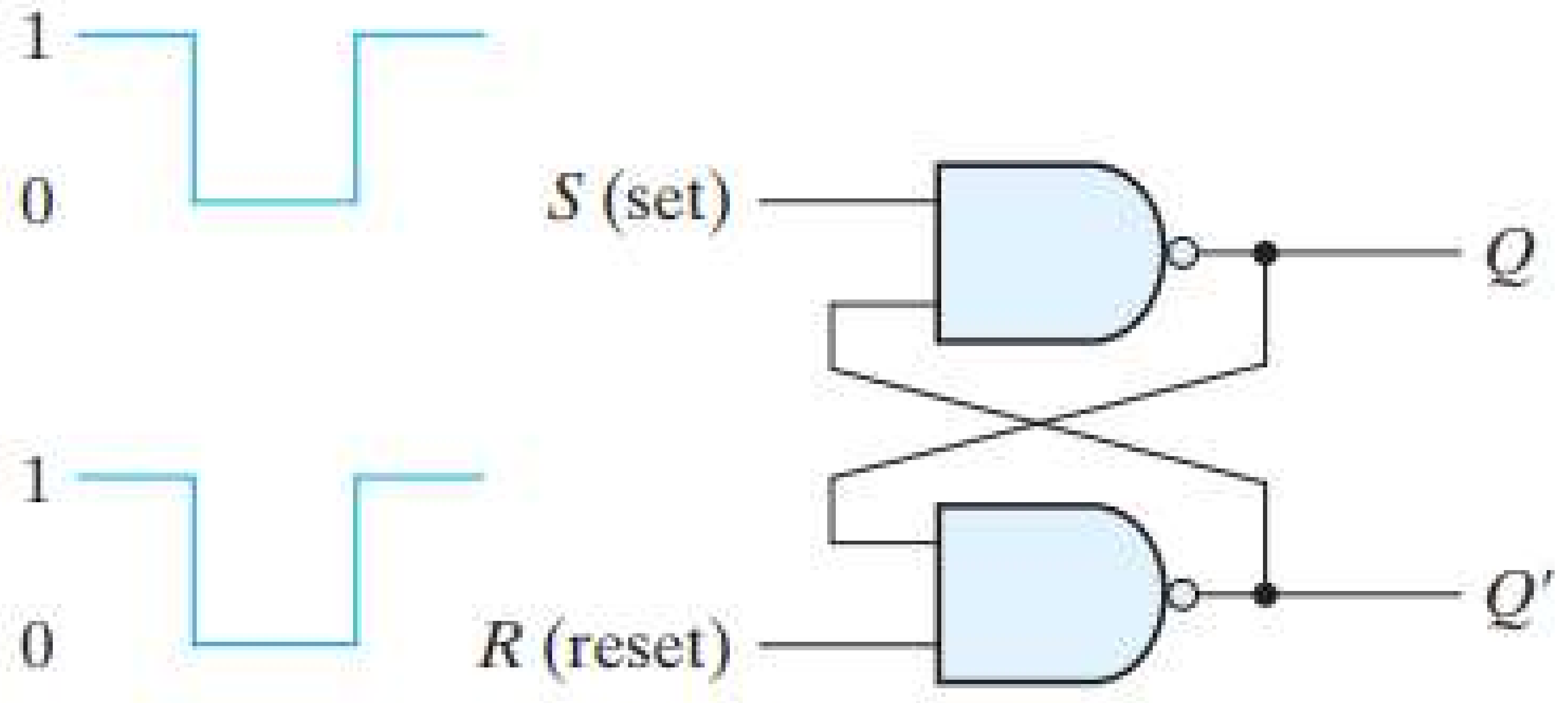
- A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states.
- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state.
- *Storage elements that operate with signal levels (rather than signal transitions) are referred to as* ***latches; those controlled by a clock transition are flip-flops*****.**

SR Latch

- The *SR* latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled *S* for set and *R* for reset.
- The latch has two useful states.
- When output $Q = 1$ and $Q' = 0$, the latch is said to be in the *set state*. When $Q = 0$ and $Q' = 1$, it is in the *reset state*. The *SR* latch constructed with two cross-coupled NOR gates is shown in the next slide.

SR Latch – Logic Diagram

The *SR* latch constructed with two cross-coupled **NOR gates** is shown below.



(a) Logic diagram

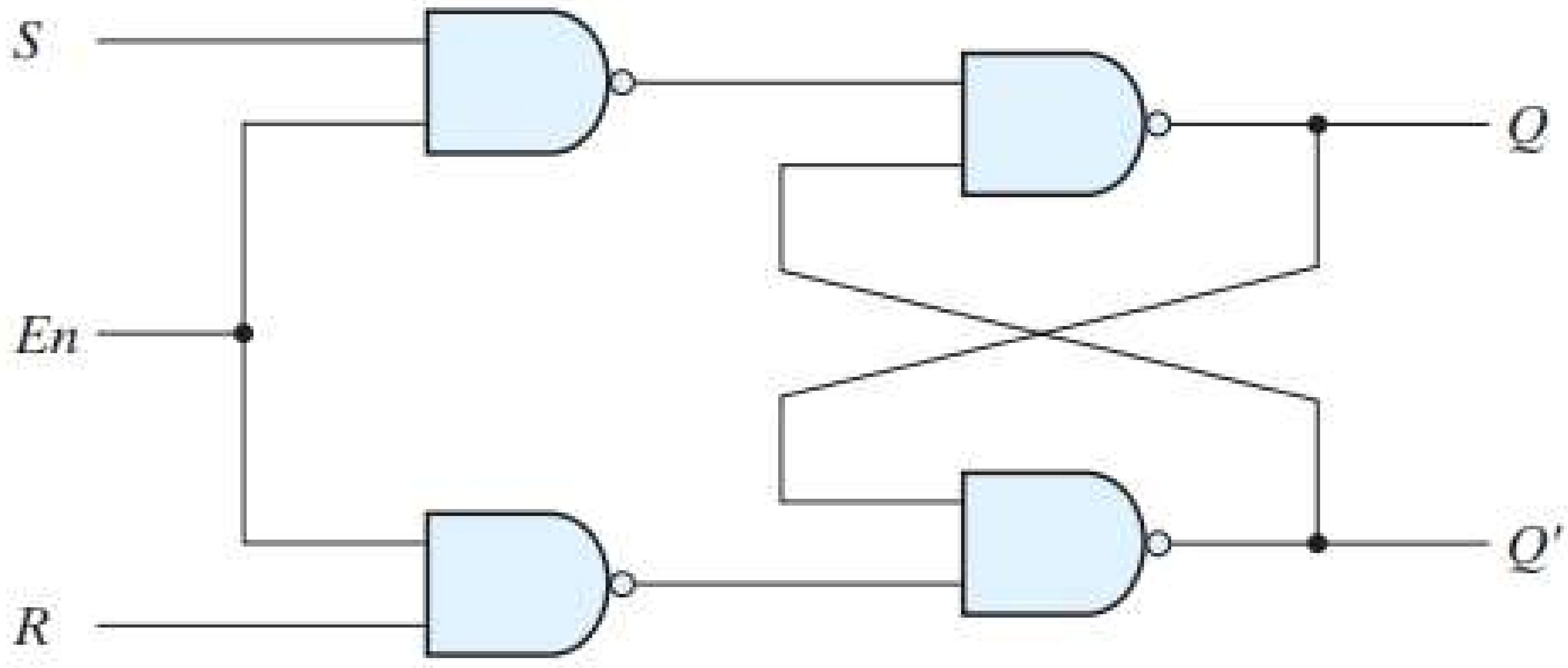
SR Latch – Functional Table

S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

(b) Function table

SR Latch – Logic Diagram

The *SR* latch with two cross-coupled **NAND gates** is shown below.



(a) Logic diagram

SR Latch – Functional Table

En	*S*	*R*	Next state of *Q*
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

SR Latch

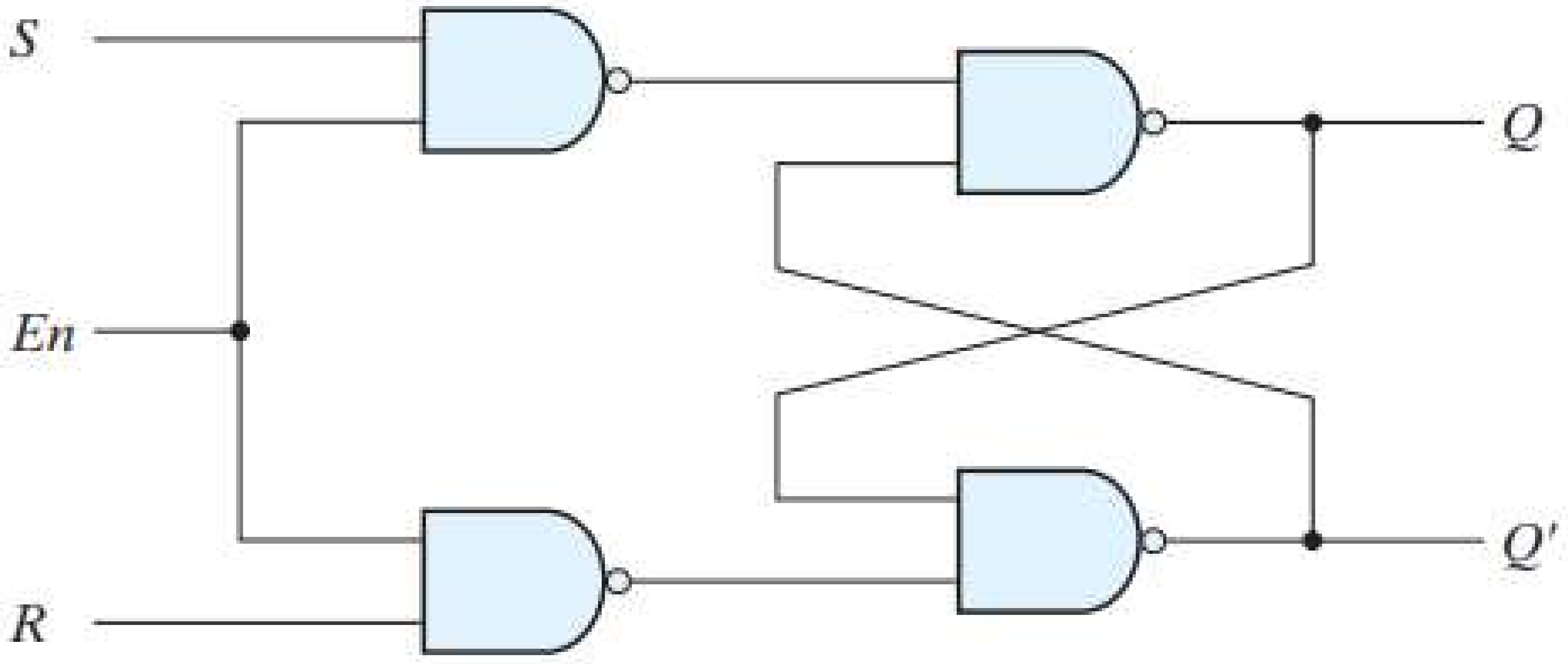
- In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an $S'R'$ latch.
- The primes (or, sometimes, bars over the letters) designate the fact that the inputs must be in their complement form to activate the circuit.

Modified SR Latch

- The operation of the basic *SR* latch can be modified by providing an additional input signal that determines (controls) *when* the state of the latch can be changed by determining whether *S* and *R* (or *S*′ and *R*′) can affect the circuit.

- It consists of the basic *SR* latch and two additional NAND gates. The control input *En* acts as an *enable* signal for the other two inputs. **The outputs of the NANDgates stay at the logic-1 level as long as the enable signal remains at 0.**

Modified SR Latch – Logic Diagram



(a) Logic diagram

Modified SR Latch – Functional Table

En	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

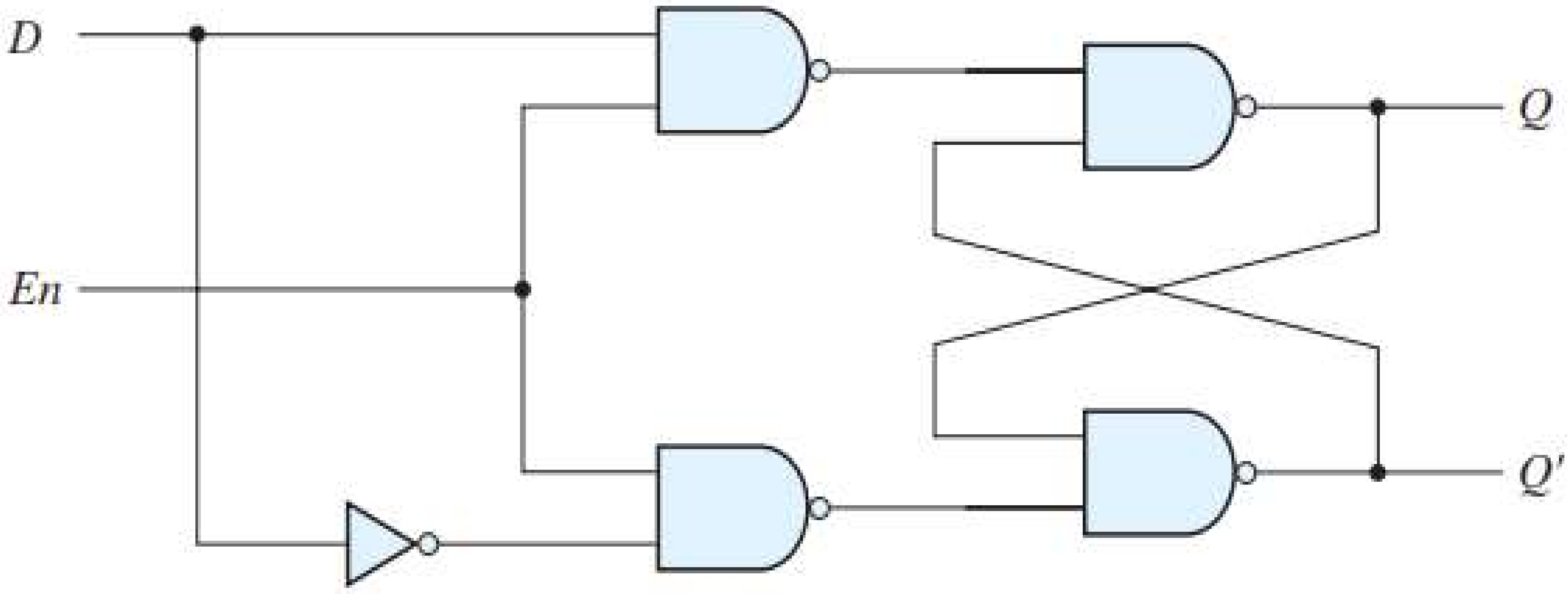
Modified SR Latch

- An **indeterminate condition** occurs when all three inputs are equal to 1. This condition places 0's on both inputs of the basic *SR* latch, which puts it in the undefined state.
- When the enable input goes back to 0, one cannot conclusively determine the next state, because it depends on whether the *S* or *R* input goes to 0 first. This indeterminate condition makes this circuit difficult to manage, and it is seldom used in practice.

D Latch

- One way to eliminate the undesirable condition of the indeterminate state in the *SR* latch is to ensure that inputs *S* and *R* are never equal to 1 at the same time. This is done in the *D* latch, shown in next slide.
- This latch has only two inputs: *D* (data) and *En* (enable).

D Latch – Logic Diagram



(a) Logic diagram

D Latch – Functional Table

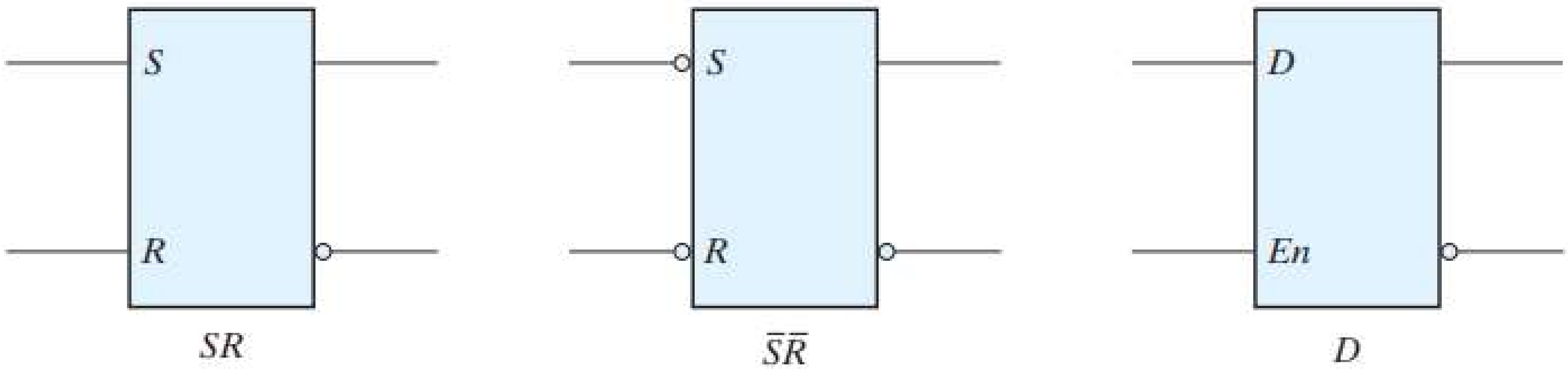
En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

(b) Function table

D Latch - Functioning

- The *D* input goes directly to the *S* input, and its complement is applied to the *R* input. As long as the enable input is at 0, the cross-coupled *SR* latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of *D*.
- The *D* input is sampled when *En* = 1. If *D* = 1, the *Q* output goes to 1, placing the circuit in the set state. If *D* = 0, output *Q* goes to 0, placing the circuit in the reset state.

Graphical Symbol for Latches



Sequential Circuits: Flip Flops

Storage Elements: Flip-Flops

- The state of a latch or flip-flop is switched by a change in the control input. This momentary change is called a ***trigger***, and the transition it causes is said to trigger the flip-flop.
- The problem with the latch is that it responds to a change in the *level* of a clock pulse.
- Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a common clock.

Storage Elements: Flip-Flops

- The key to the proper operation of a flip-flop is to trigger it only during a signal *transition*.
- A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0.
- The positive transition is defined as the **positive edge** and the negative transition as the **negative edge** as shown in the next slide.

Storage Elements: Flip-Flops

(a) Response to positive level

(b) Positive-edge response

(c) Negative-edge response

Storage Elements: Flip-Flops

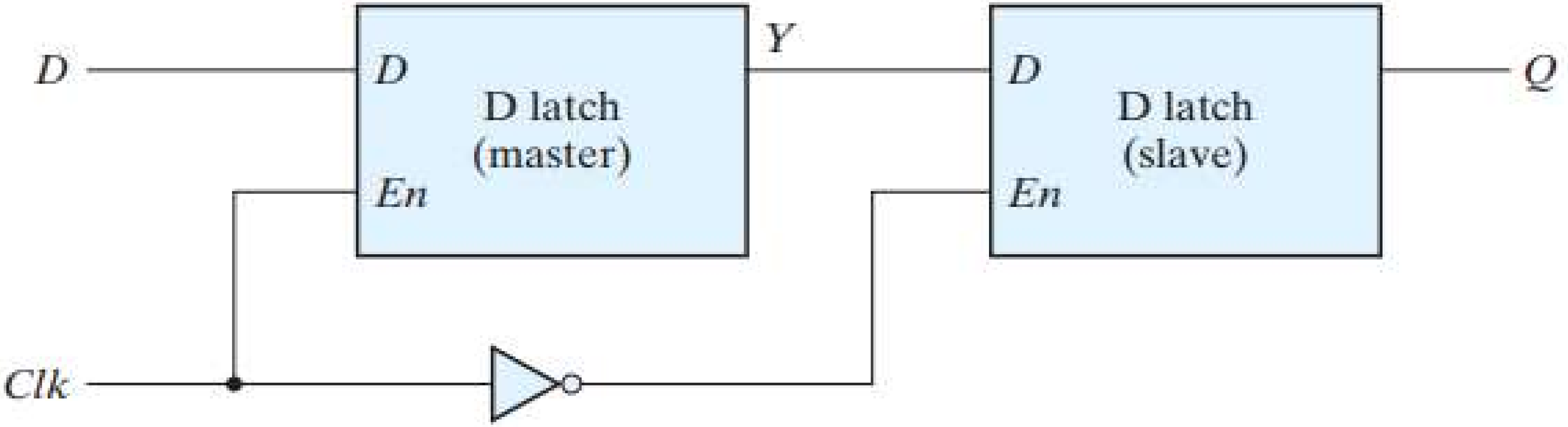
There are two ways that a latch can be modified to form a flip-flop.

1. One way is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing.

2. Another way is to produce a flip-flop that triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing signal (clock) and is disabled during the rest of the clock pulse.

Edge Triggered D Flip-Flop

The construction of a *D* flip-flop with two *D* latches and an inverter is shown below. The first latch is called the master and the second the slave.



Edge Triggered D Flip-Flop-Working

- The circuit samples the *D* input and changes its output *Q* only at the negative edge of the synchronizing or controlling clock (designated as *Clk*). When the clock is 0, the output of the inverter is 1.
- The slave latch is enabled, and its output *Q* is equal to the master output *Y*. The master latch is disabled because *Clk* = 0.
- When the input pulse changes to the logic-1 level, the data from the external *D* input are transferred to the master.

Edge Triggered D Flip-Flop

- The slave, however, is disabled as long as the clock remains at the 1 level, because its *enable* input is equal to 0. Any change in the input changes the master output at *Y*, but cannot affect the slave output.
- When the clock pulse returns to 0, the master is disabled and is isolated from the *D* input. At the same time, the slave is enabled and the value of *Y* is transferred to the output of the flip-flop at *Q*. Thus, *a change in the output of the flip-flop can be triggered only by and during the transition of the clock from 1* to 0.

Edge Triggered D Flip-Flop

The behavior of the master–slave flip-flop just described dictates that

(1) the output may change only once,

(2) a change in the output is triggered by the negative edge of the clock, and

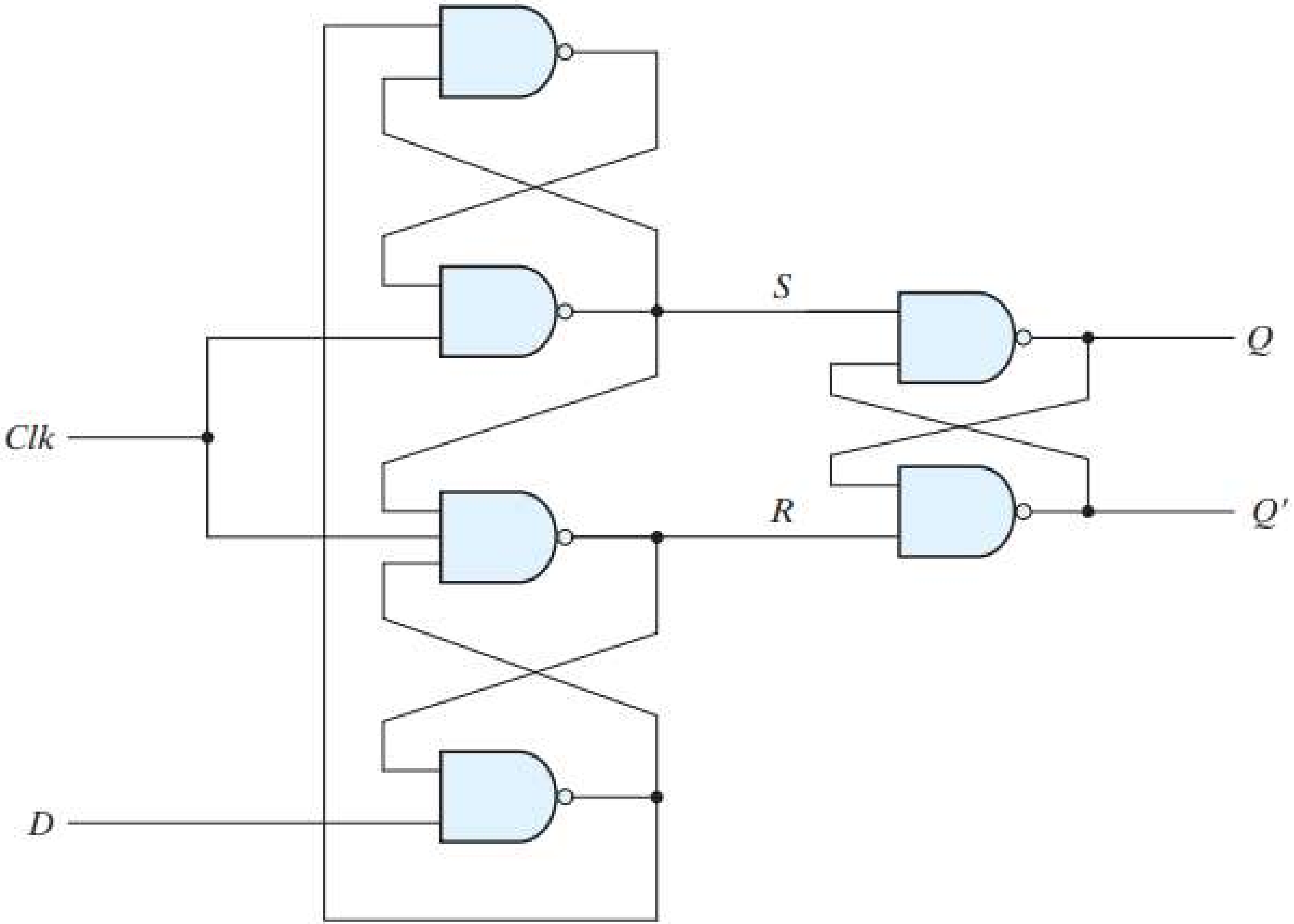
(3) the change may occur only during the clock’s negative level.

Edge Triggered D Flip-Flop

Another construction of an edge-triggered *D* flip-flop uses three *SR* latches as shown next slide.

- Two latches respond to the external *D* (data) and *Clk* (clock) inputs. The third latch provides the outputs for the flip-flop.

Edge Triggered D Flip-Flop



Edge Triggered D Flip-Flop - Working

- The *S* and *R* inputs of the output latch are maintained at the logic-1 level when *Clk* = 0. This causes the output to remain in its present state.
- Input *D* may be equal to 0 or 1. If *D* = 0 when *Clk* becomes 1, *R* changes to 0. This causes the flip-flop to go to the reset state, making *Q* = 0. If there is a change in the *D* input while *Clk* = 1, terminal *R* remains at 0 because *Q* is 0.

Edge Triggered D Flip-Flop

Thus, the flip-flop is locked out and is unresponsive to further changes in the input. When the clock returns to 0, *R* goes to 1, placing the output latch in the quiescent condition without changing the output. Similarly, if *D* = 1 when *Clk* goes from 0 to 1, *S* changes to 0.

This causes the circuit to go to the set state, making *Q* = 1. Any change in *D* while *Clk* = 1 does not affect the output.

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 12

Synchronous Sequential Logic

Weekly Learning Outcomes

1. Describe a sequential circuit in terms of a state equation
2. Learn how to write a state table for a sequential circuit
3. Learn how to draw a state diagram for a sequential circuit
4. Learn how to simplify a sequential circuit design to reduce the number of logic gates needed
5. Learn the design process for a sequential circuit using different sequential circuit building blocks (D flip-flops, JK flip-flops, T flip-flops)

Required Reading

1. Chapter 5 (Sections 5.5, 5.7, 5.8) (Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapters 17 (John Seiffertt Digital Logic for Computing)
2. https://www.youtube.com/watch?v=HXG_YPVNIsM

Analysis vs. design

- The **analysis** of a sequential circuit **starts** from a **circuit diagram** and finishes with a **state table** or a state **diagram**

- The **design** of a sequential circuit **starts** from a set of **specifications** from which a **state diagram** is obtained and finally the **circuit diagram**

Characteristic tables

- Sequential circuits typically use one of the flip-flop types that we discussed in a previous lecture
- To analyze the behavior of sequential circuits we will need the characteristic tables of these flip-flops

***JK* Flip-Flop**

J	K	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

***D* Flip-Flop**

D	$Q(t+1)$	
0	0	Reset
1	1	Set

***T* Flip-Flop**

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

- Recall Characteristic Tables: **<u>given the input</u>** **<u>determine the next state</u>**

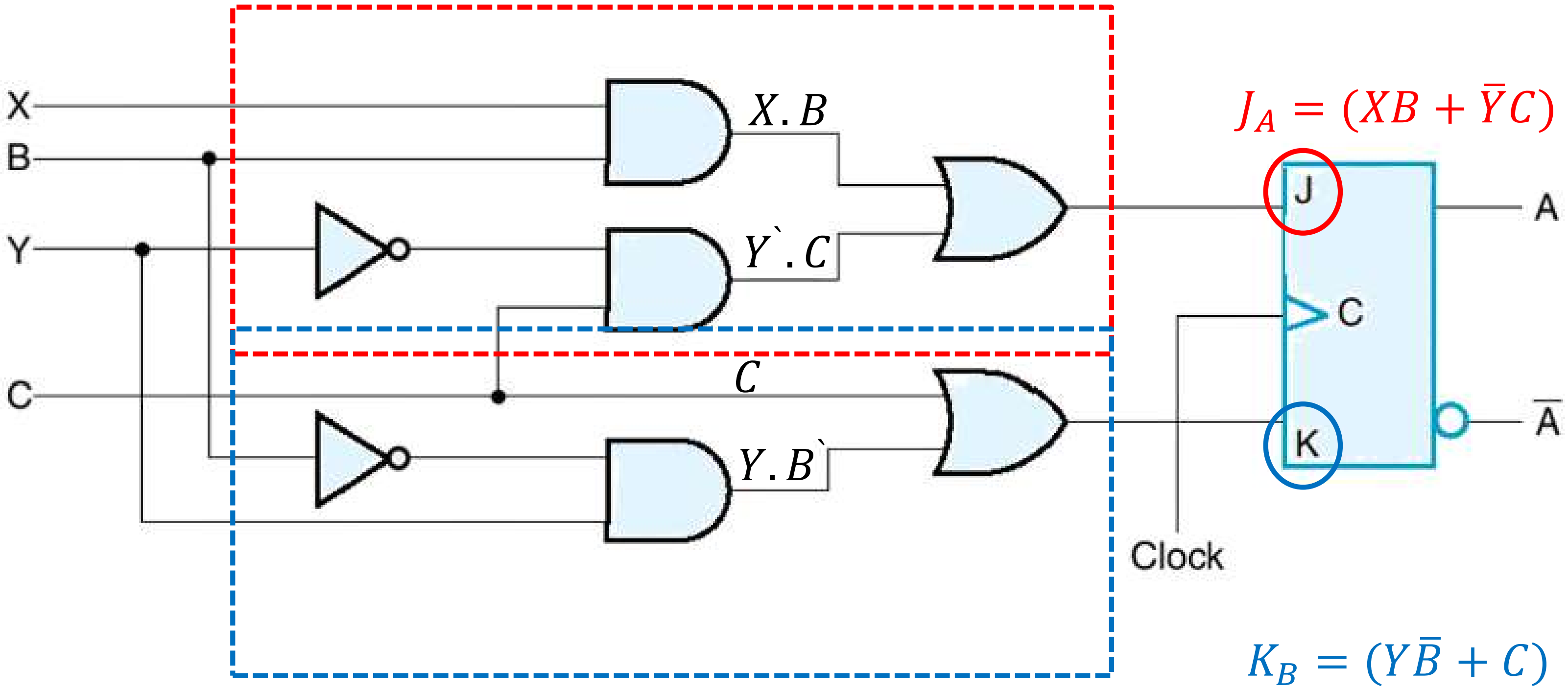
Analysis of Sequential circuits

- The behavior of a sequential circuit is determined by:
  - Inputs
  - Present state of the circuit
  - Outputs
- The analysis of a sequential circuit consists of :
  - Deriving the Input Equations to the Flip-Flops
  - Getting the State Table
  - Obtaining the State Diagram, which demonstrates the time sequence of inputs, outputs, and states

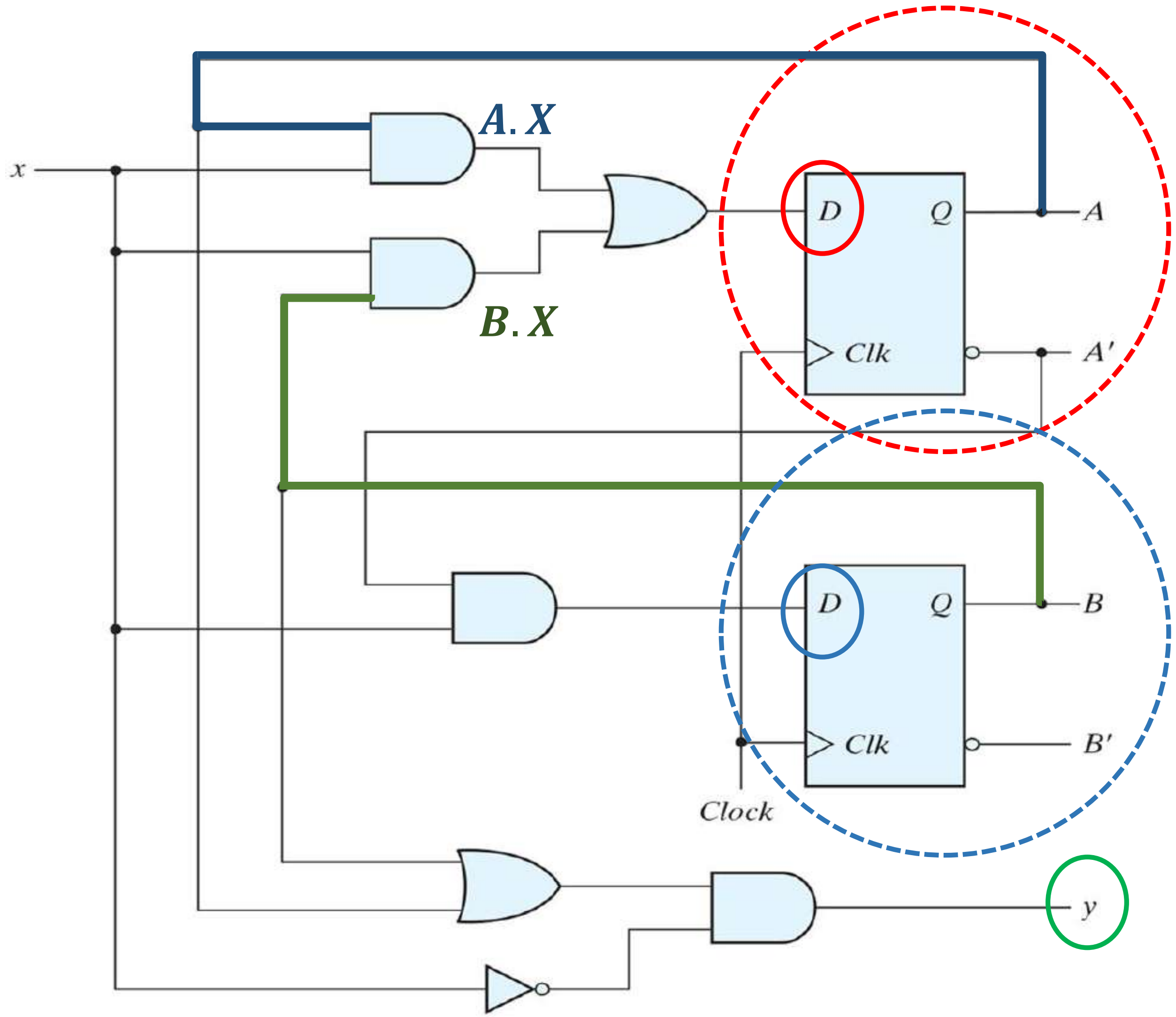
Analysis example

Step 1: Derive Input Equations

- Describe the inputs to the flip-flop using logic equations



Another Example (Derive Input Equations)



$D_A = (AX + BX)$

$D_B = (\bar{A}X)$

$Y = (A + B)\bar{X}$

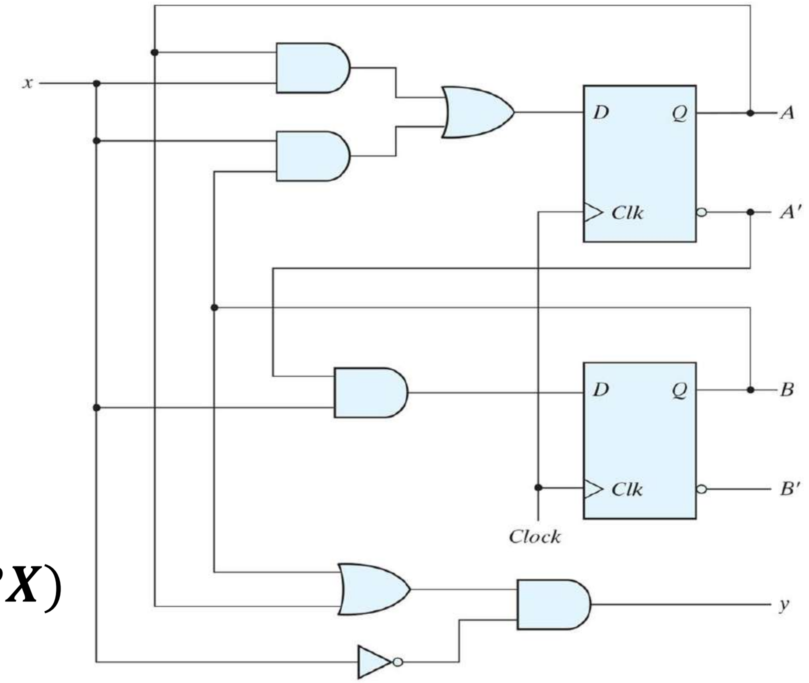
Analysis: Input Equation

Input Equation

- The input equation
  - Imply the **type of flip-flop** from the letter symbols
  - Fully specify **the combinational circuit** that drives the flip flops

Input Equation

- Note that in the previous circuit to **determine the next state** it used
  - Present state (A, B, …)
  - And the input (X)
- To determine the **output,** it also used
  - Present state (A, B, …)
  - And the input (X) (complemented)
- Synchronous circuit (clock driven)



$$D_A = (AX + BX)$$

$$D_B = (\bar{A}X)$$

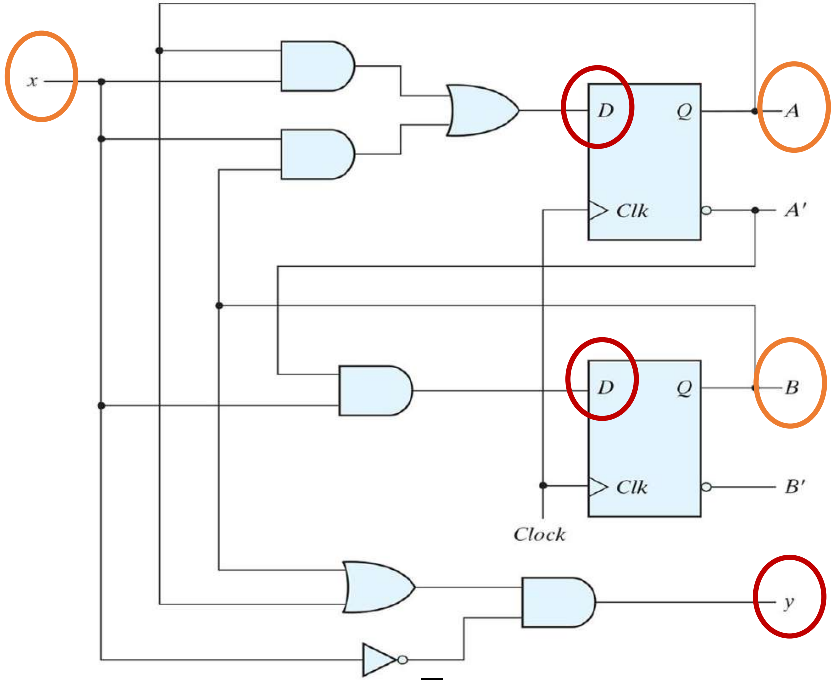
$$Y = (A + B)\bar{X}$$

Analysis: State Table

Step 2: State Table

- Like the truth table but with state added
  - A sequential circuit with m FFs and n inputs needs 2^{m+n} rows
  - Next state and output are determined using **Input Equations**

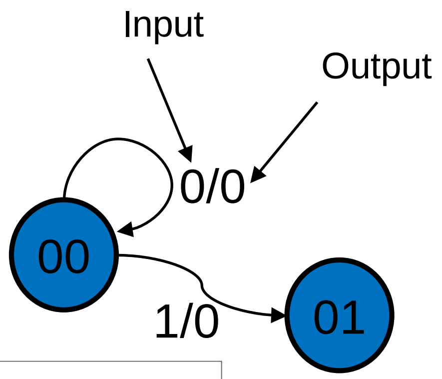
Left side (input)			Right side (State, Output)		
Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

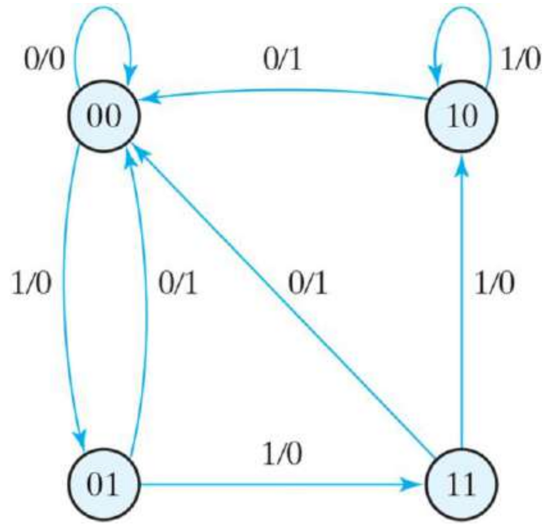


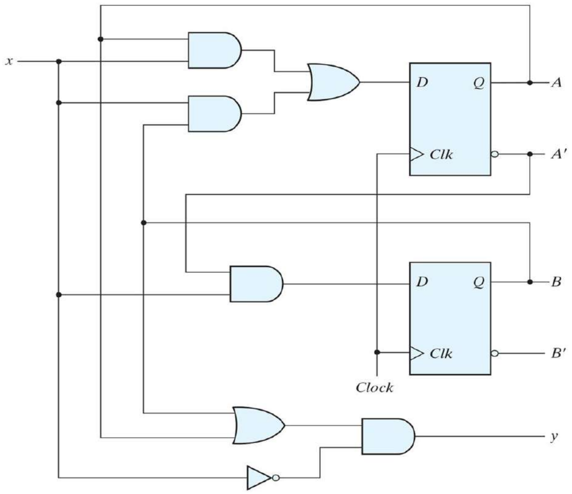
$D_A = (AX + BX)$ $D_B = (\bar{A}X)$ $Y = (A + B)\bar{X}$

Step 3: State Diagram (Mealy Model)

- An alternative representation to the state table



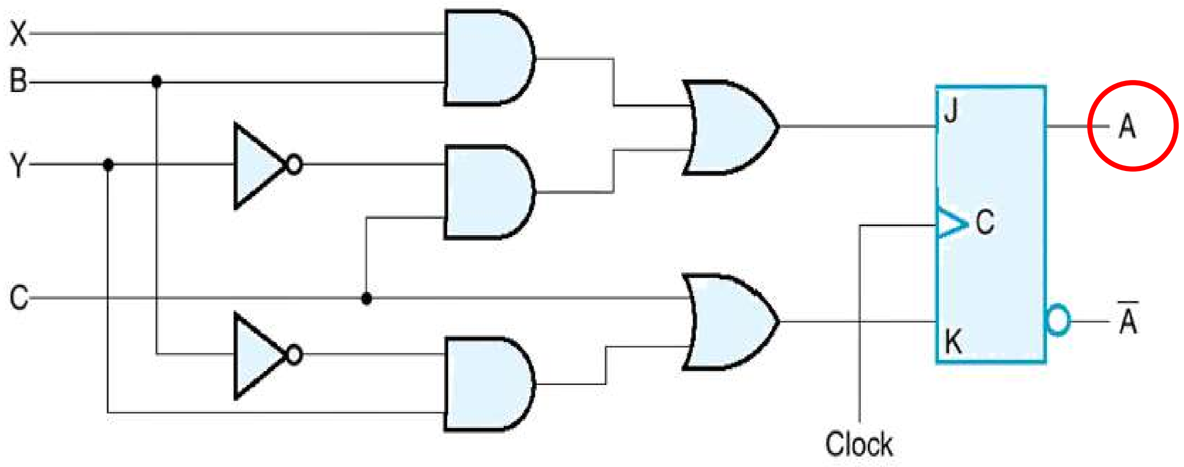


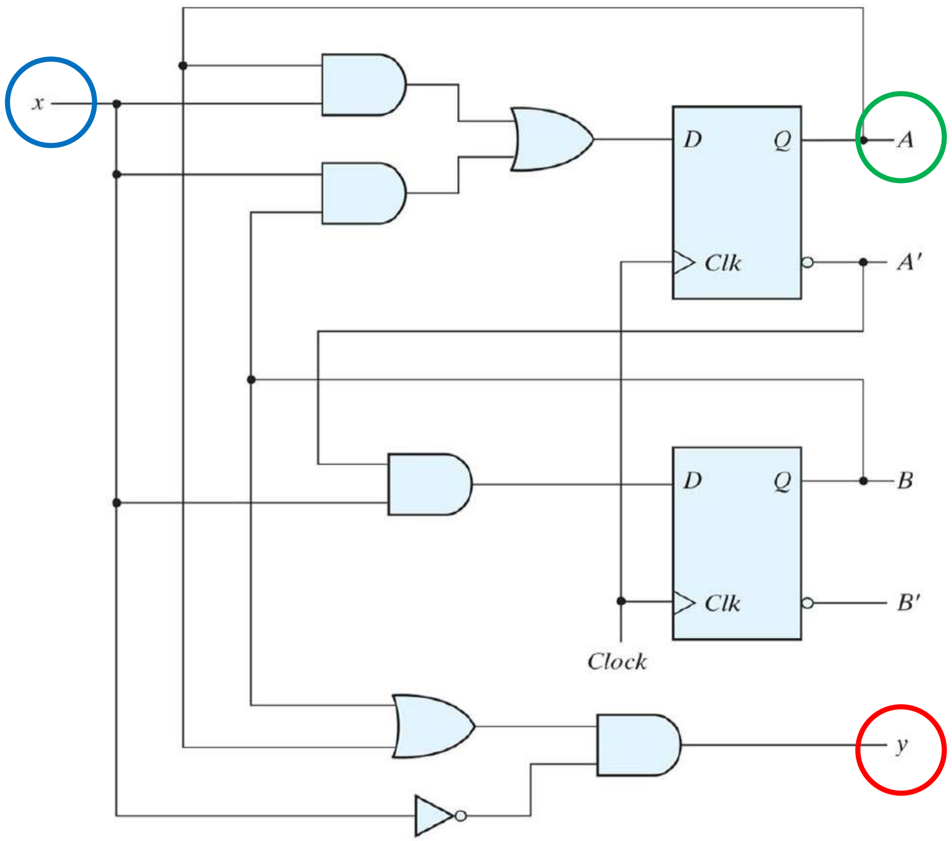


Present State		Input	Next State		Output
A	***B***	***x***	***A***	***B***	***y***
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Mealy Model – outputs depends on **inputs and States**

Sequential Circuit Types



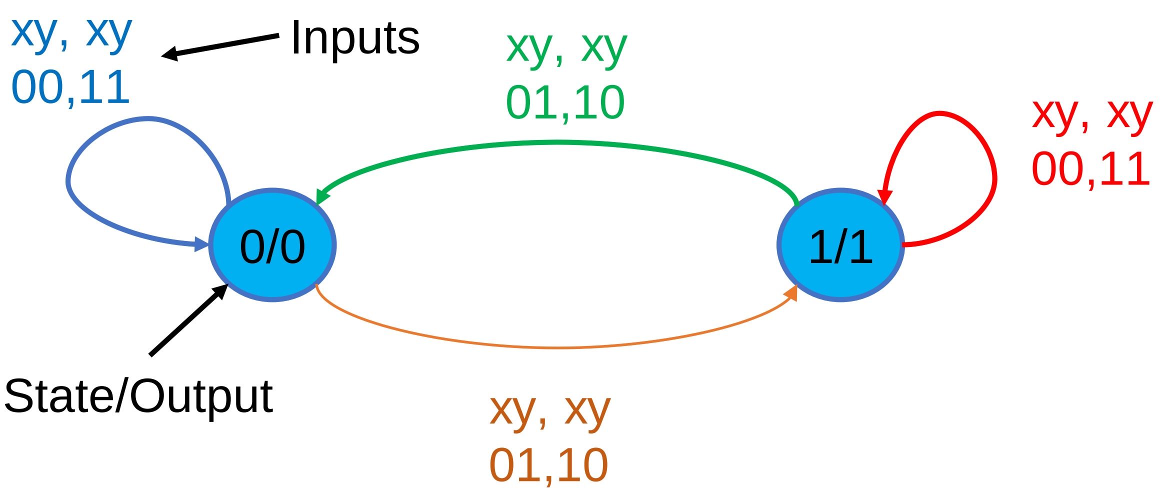


Moore Model – outputs depends on states only

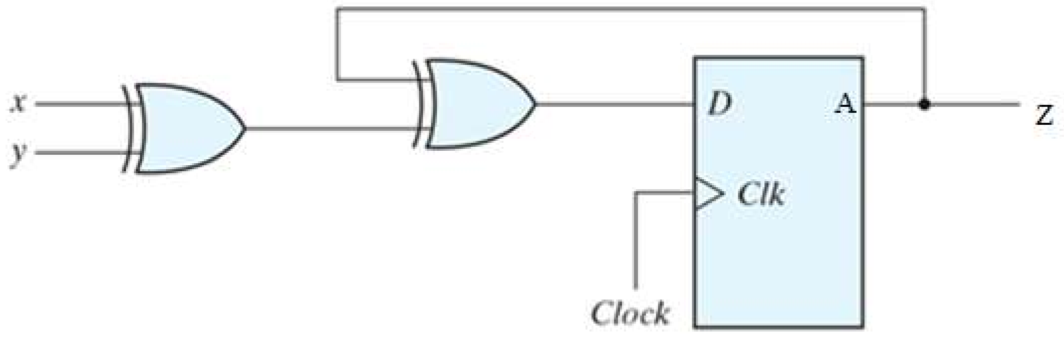
Mealy Model – outputs depends on inputs & states

$$Y = (A + B)\bar{X}$$

State Diagram : More



Present state	Inputs		Next state	Output
A	x	y	A	Z
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Moor vs. Mealy Machine

- **Moore Machine:**
  - Easy to understand and easy to implement/code
  - It may require more states (more hardware)
- **Mealy Machine**
  - More complex because the outputs are functions of both the state and the input
  - Requires less states in most casts (less hardware components)
- The choice of a mode l depends on the application and personal preference

State Table vs. State Diagram

- The provide the same information
  - **Table** is easier to fill from the problem description
  - **Diagram** is easier to understand

Analysis: Various flip-flops

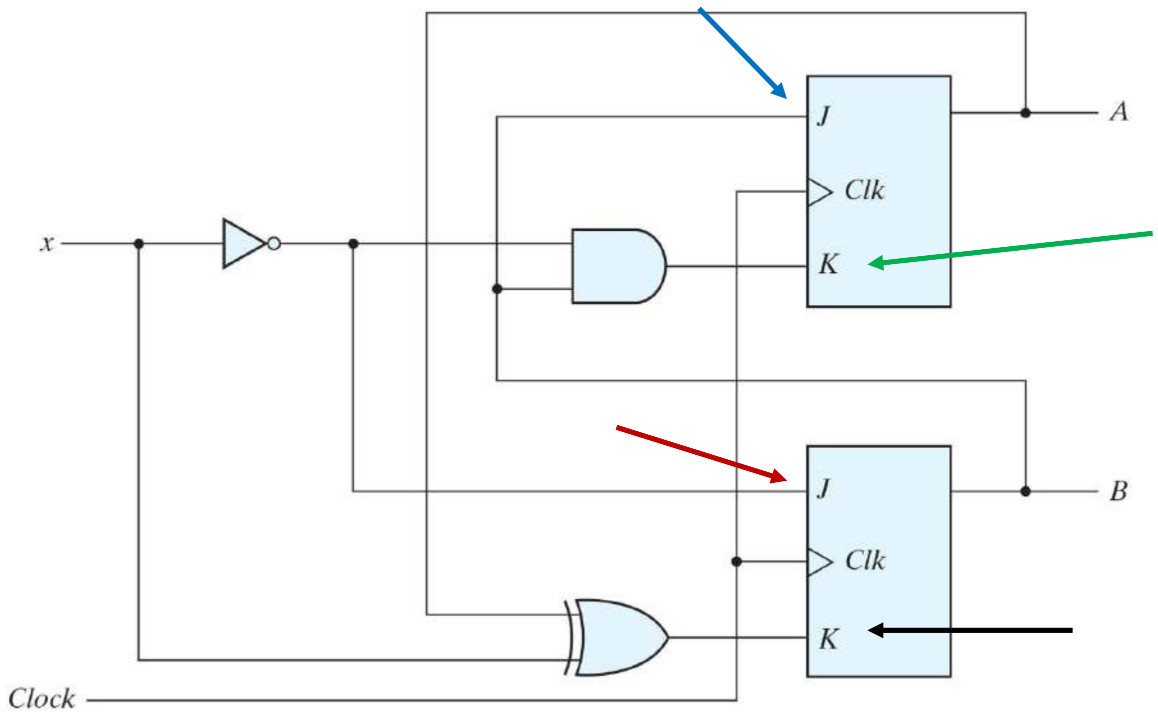
Analysis with D Flip-Flops

- Analysis for sequential circuits using D flip flops are easier. Why?
  - Because the next state is obtained directly from the input equations

D Flip-Flop

D	$Q(t+1)$	
0	0	Reset
1	1	Set

Analysis with JK Flip-Flops



- $J_A = B$
- $J_B = x^`$
- $K_A = Bx^`$
- $K_B = A^` x + Ax^` = A \oplus x$

- How Many entries do we have in the table?

Analysis with JK Flip-Flops (State Table)

Present State		Input	Next State		Flip-Flop Inputs			
A	*B*	*x*						
0	0	0						
0	0	1						
0	1	0						
0	1	1						
1	0	0						
1	0	1						
1	1	0						
1	1	1						

- $J_A = B$
- $J_B = x^{`}$
- $K_A = Bx^{`}$
- $K_B = A^{`}x + Ax^{`} = A \oplus x$

- Input equations determine the FF inputs
- Present state, and J & K values determine the next state (see the characteristic table)

Characteristic Table of JK FF

***JK* Flip-Flop**			
J	***K***	**Q(*t* + 1)**	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

Analysis with JK Flip-Flops (State Table)

Present State		Input	Next State		Flip-Flop Inputs			
A	***B***	***x***	***A***	***B***	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

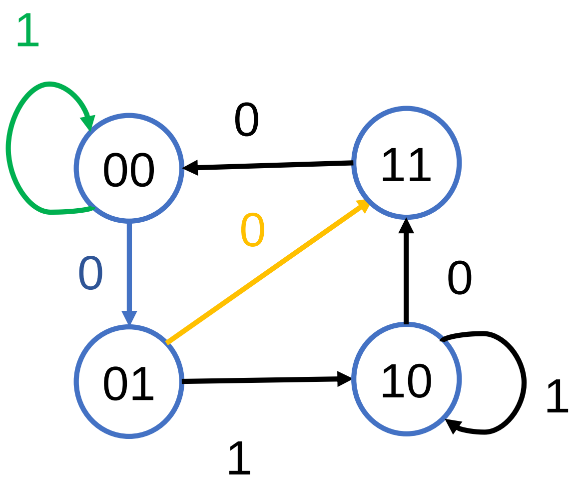
- Input equations determine the FF inputs
- Present state, and J & K values determine the next state (see the characteristic table)

- $J_A = B$
- $J_B = x^`$
- $K_A = Bx^`$
- $K_B = A^`x + Ax^` = A \oplus x$

Characteristic Table of JK FF

***JK* Flip-Flop**			
J	***K***	***Q(t + 1)***	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

Analysis with JK Flip-Flops (State Diagram)



Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

Designing circuits

Analysis vs. Design

- The **analysis** of a sequential circuit **starts** from a **circuit diagram** and finishes with a **state table** or a state **diagram**

- The **design** of a sequential circuit **starts** from a set of **specifications** from which a **state diagram** is obtained and finally the **circuit diagram** (with logic gates)

Design procedure

- Design starts from **a specification** and results in a logic diagram or a list of Boolean functions
- The **steps** are :
  - Derive a **state diagram**
    - Assign binary values to the states
  - Choose the **type of flip flop** to use
  - Obtain the **state table**
    - Utilizing the characteristic tables of the flip-flop
  - Get the **simplified input equations** and output equations
  - Draw the **logic diagram**

Sequential Circuit Design

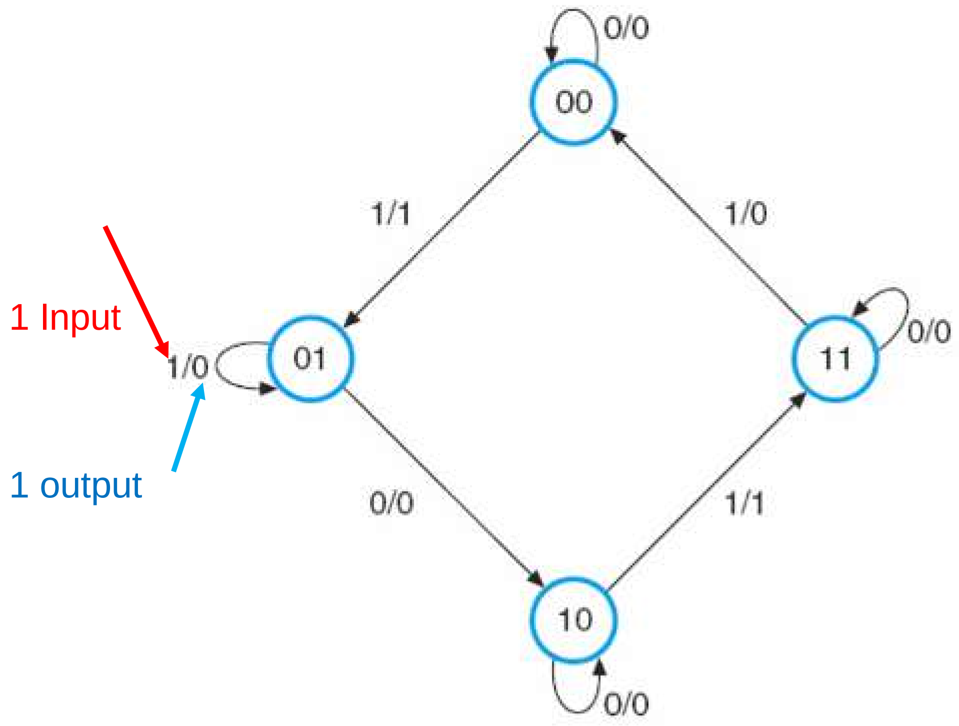
- Synchronous sequential circuits are made up of flip flops and combinational gates.
- Part of the design process is choosing the flip-flop type and the combinational circuit structure to meet the specification
- How many flip flops?
  - The number of flip-flops depends on the number of states
  - N flip-flops can represent up to 2^n binary states
  - For example
    - 2 states requires one flip flop
    - 4 states requires two flip flops
    - 7 states requires three flop flops

Design with D Flip Flops

Designing with D Flip Flops

- Four States
- We need $\log_2(4) = 2$ flip flops

- Design a clocked sequential circuit that implement this state diagram
- Using D flip flops



Designing with D Flip Flops

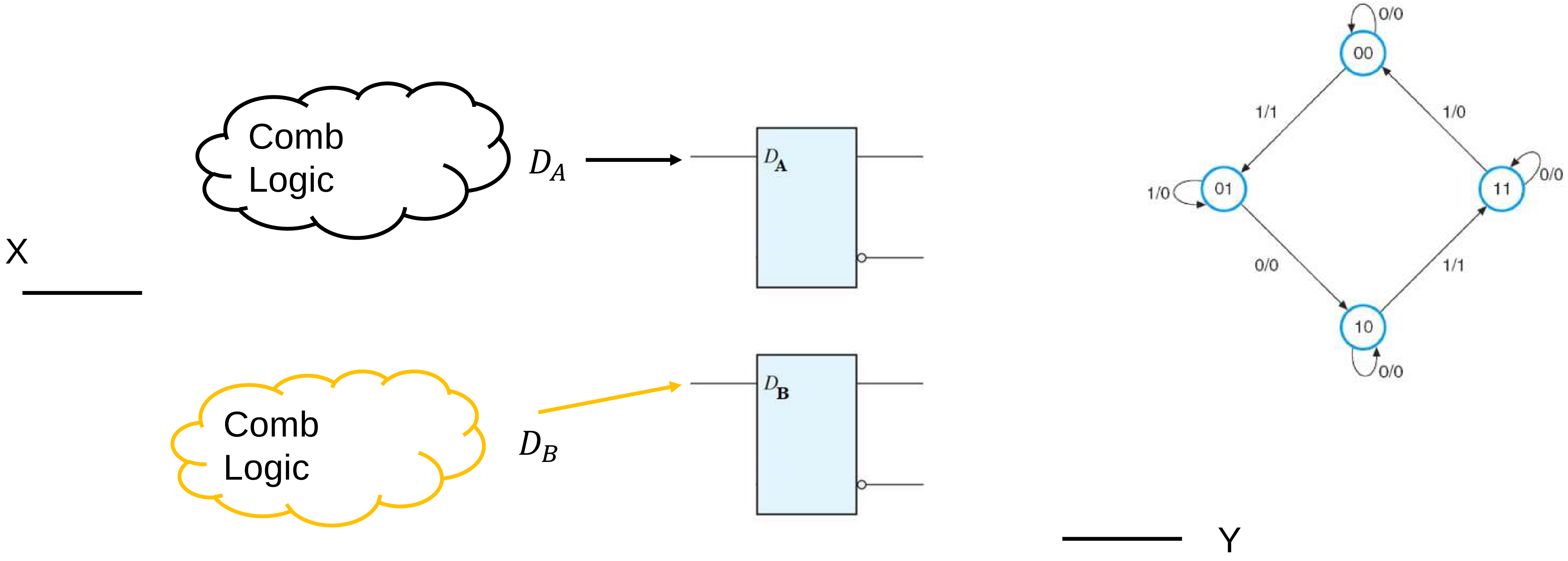
- Now we select tow D flip flops representing the four states and we label their outputs as A and B
- We have **one input** that we label as **x** and **one output** that we label as **y**
- Remember that the characteristic equation of the D flip flop is given by
    - $Q(t+1) = D_Q$ (next state value equals current state input)

D Flip-Flop

D	$Q(t+1)$	
0	0	Reset
1	1	Set

Designing with D Flip Flops

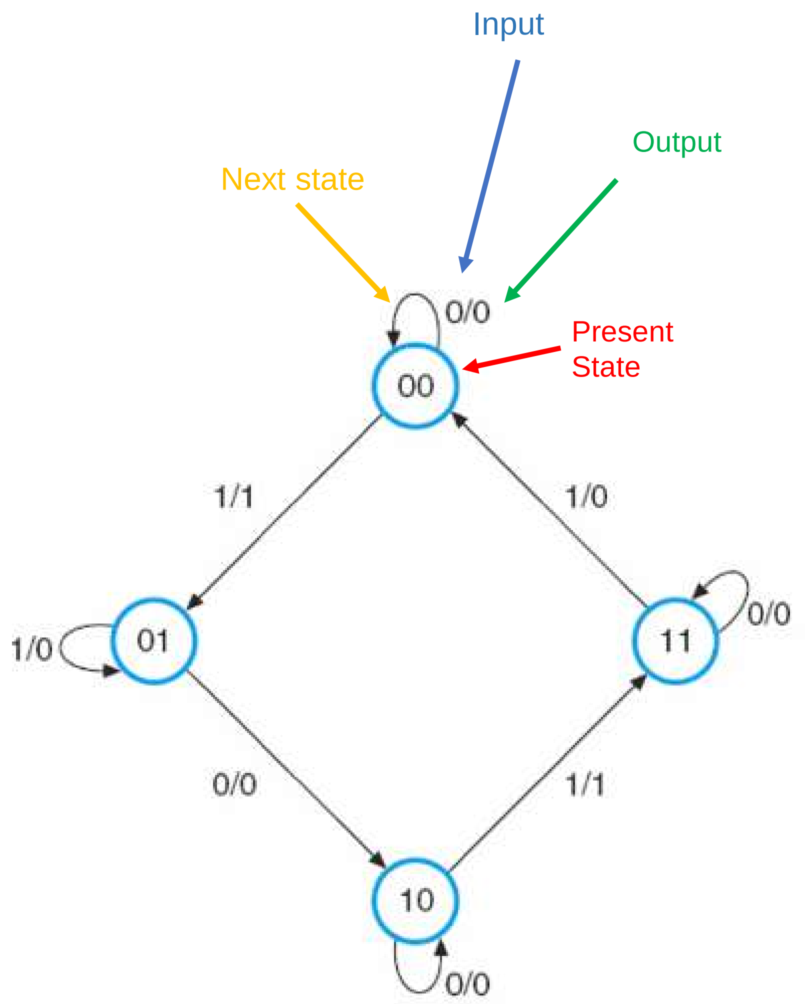
Determine the logic that drives the flip flop



Designing with D Flip Flops

- Input equation can be obtained directly from the table using minterms:
  - $A(t+1) = D_A(A,B,x) = \Sigma m(2,4,5,6)$
  - $B(t+1) = D_B(A,B,x) = \Sigma m(1,3,5,6)$
  - $Y(t+1) = \Sigma m(1,5)$

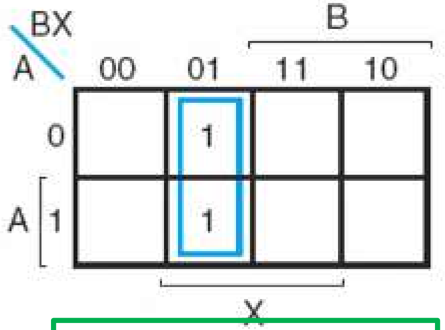
Present State		Input	Next State		Output
A	B	X	A	B	Y
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0



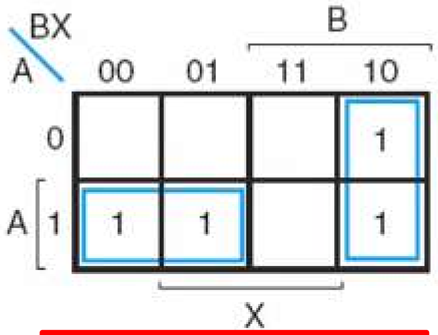
Designing with D Flip Flops

- We can use **K-Maps** to **minimize** the expressions

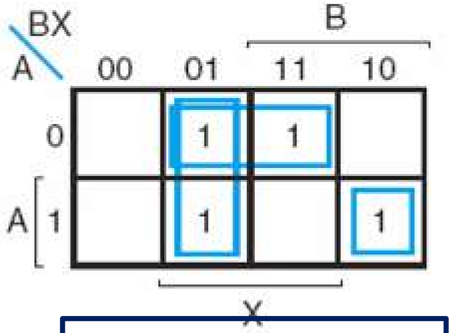
Present State		Input	Next State		Output
A	B	X	A	B	Y
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0



$Y = \bar{B}X$



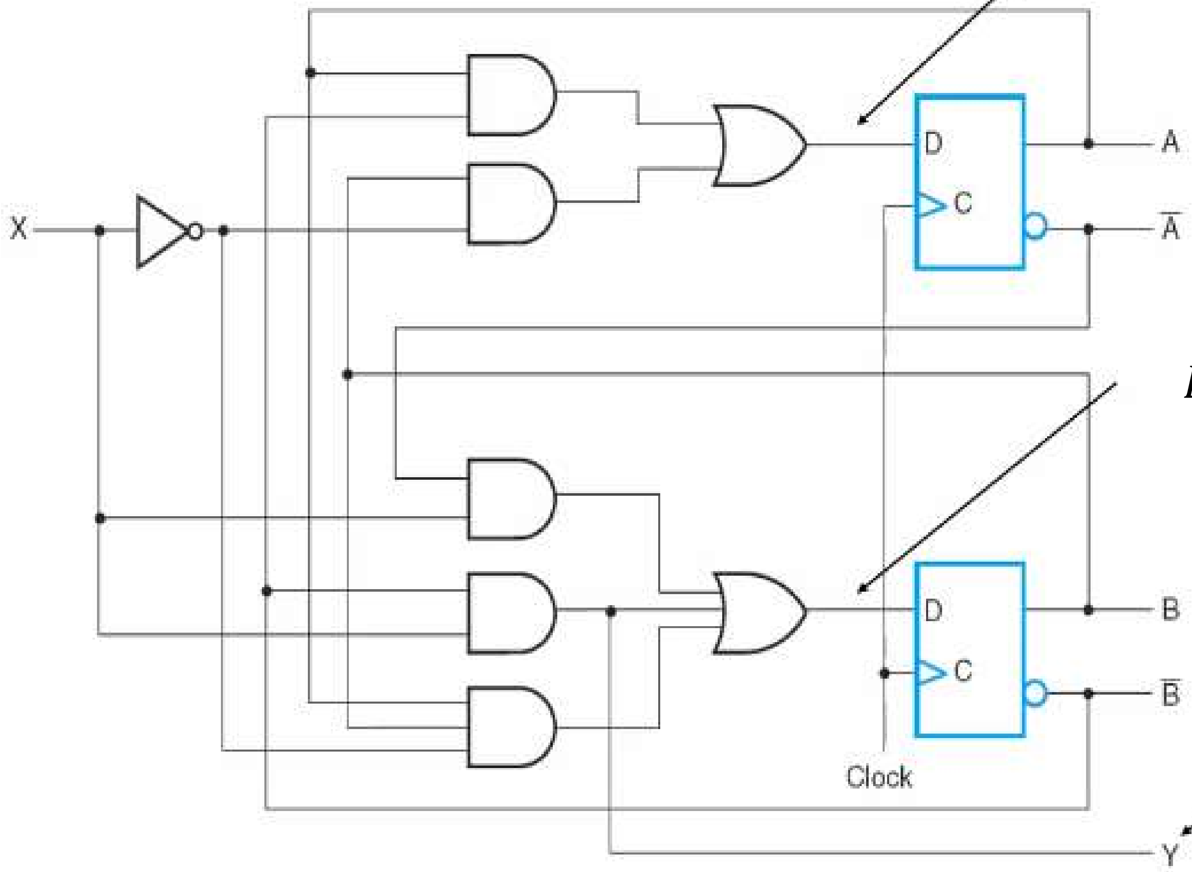
$D_A = A\bar{B} + B\bar{X}$



$D_B = \bar{A}X + \bar{B}X + AB\bar{X}$

Designing with D Flip Flops

$D_A = AB^{`} + BX^{`}$



$D_B = A^{`}X + B^{`}X + ABX^{`}$

$Y = B^{`}X$

- Design with T Flip Flops
- For JK Flip-Flop example refer to the link in the recommended reading (https://www.youtube.com/watch?v=HXG_YPVNIsM)

Sequential Circuits with Different Flip Flops

- Designing sequential circuits with flip flops other than D-Flip-Flops is complicated
- The input equations are derived indirectly from the state table
- It is necessary to derive a functional relationship between the state table and the input equations

Excitation Tables

- During the design process, we usually know the transition from present to next state, but we need to find the flip flop input conditions that will cause this transition
- We need a table that list the required inputs for a given change in state
  - This is called an ***excitation tables***

Excitation Tables

Given inputs determine Q(t+1)
Characteristic Table

JK Flip-Flop

J	K	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

T Flip-Flop

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

Given Q(t) Q(t+1) determine inputs
Excitation Table

$Q(t)$	$Q(t = 1)$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) *JK* Flip-Flop

$Q(t)$	$Q(t = 1)$	T
0	0	0
0	1	1
1	0	1
1	1	0

(b) *T* Flip-Flop

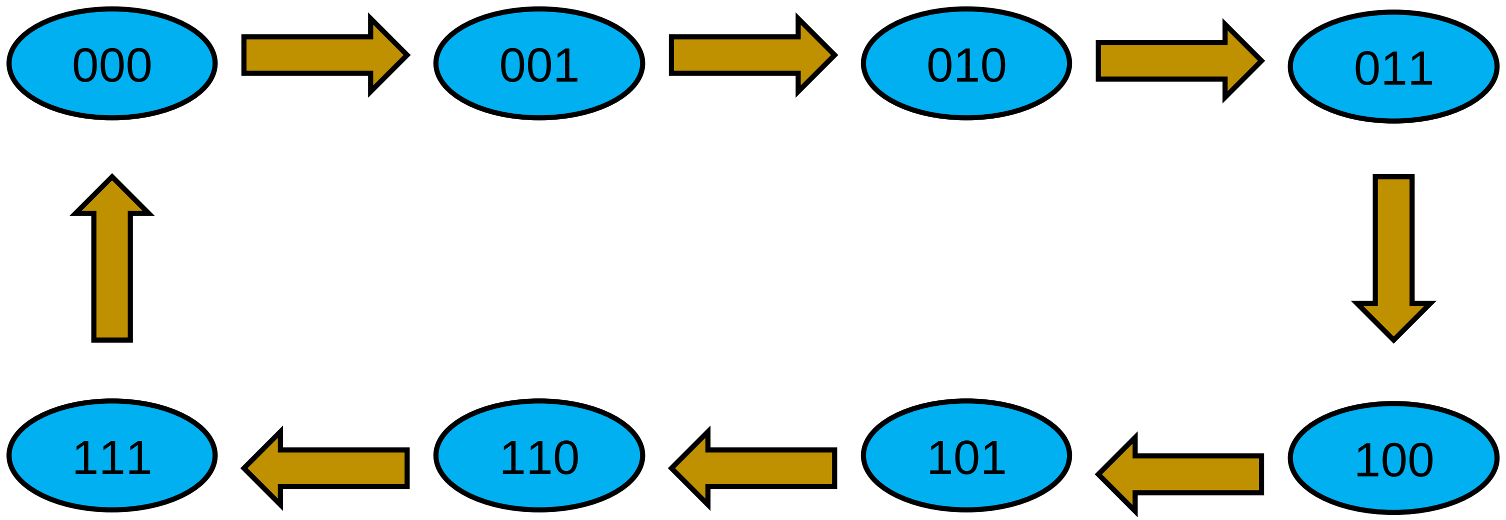
Designing with T Flip Flops

- Designing a circuits with T flip flops is the same as with D flip flops
  - Except that the input equations must be evaluated from the present-state to next state transition derived from the excitation table of the T Flip Flop

Designing A counter with T Flip Flops

- Design a counter that counts from “000” to “111” and then back to “000” again
  - Using T Flip-Flops

Designing A counter with T Flip Flops



Notice that the only input is the clock

Designing A counter with T Flip Flops

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1			
0	0	1	0	1	0			
0	1	0	0	1	1			
0	1	1	1	0	0			
1	0	0	1	0	1			
1	0	1	1	1	0			
1	1	0	1	1	1			
1	1	1	0	0	0			

$Q(t)$	$Q(t = 1)$	T
0	0	0
0	1	1
1	0	1
1	1	0

(b) *T* Flip-Flop

Designing A counter with T Flip Flops

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

$$T_{A2} = \Sigma m(3,7)$$

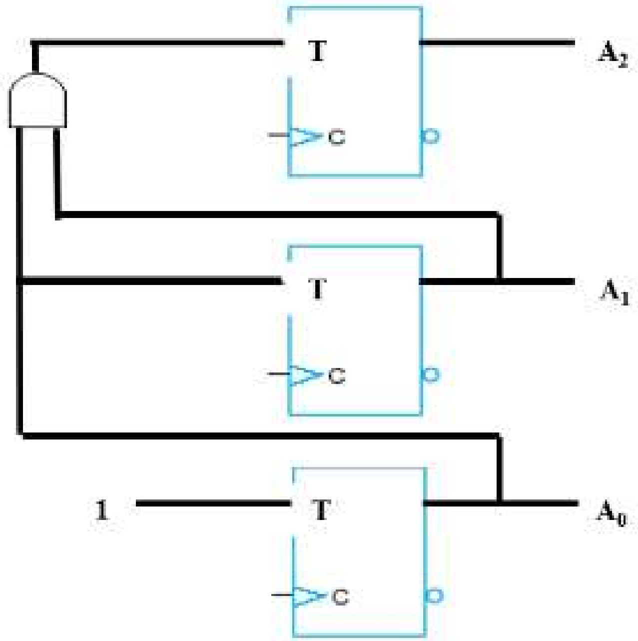
$$T_{A1} = \Sigma m(1,3,5,7)$$

$$T_{A3} = Logic\ 1\ (\text{Alwyas on, conneced to VCC})$$

We can use K-Maps to further simplify the logic

Designing A counter with T Flip Flops

- After simplification of the input equations, we get this logic diagram



State Reduction

State reduction and assignment

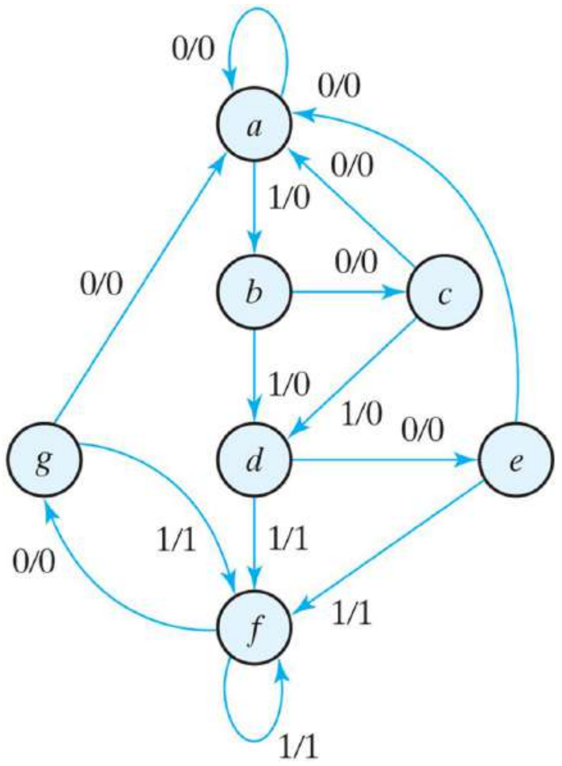
- When designing a sequential circuit, we start from a specification and end up with logic diagram
- Two sequential circuits may exhibit the same input-output relationship but have totally different number of states in the state diagram
  - Remember more states = more hardware components
- If we can reduce the number of states and keep the input-output relationship we can save in hardware cost

State reduction

- Reducing the number of flip-flops referred to as **state-reduction**
- State reduction algorithms are concerned with reducing the number of states while maintaining input-output relationship
- m flip-flops can represent 2^m states
  - Reducing the number of states does not necessarily reduce m
  - Example:
    - 4 states need 2 flip-flops
    - 5, 6, 7, and 8 states all require 3 flip-flops

State reduction example

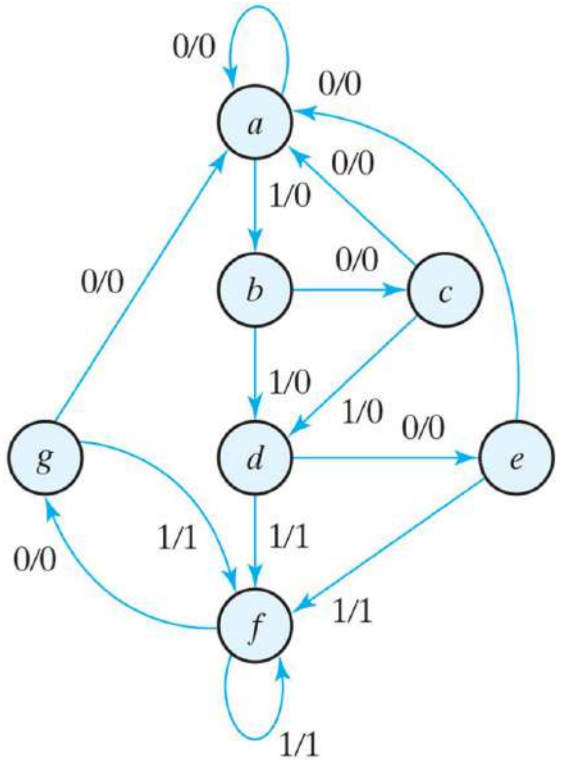
- Giving a sequential circuit defined in this state diagram we need to reduce the number of states
- We are concerned about the input/output relationship
- Internal states are used merely to provide the required sequences
  - Hence only letters are assigned to them instead of binary representation



State reduction example

- Unlike counters where the state is the same as the output in this circuit an infinite number of sequence can occur
- For example, starting in state **a** and having the input sequence **01010110100** we will end up with this sequence

state	*a*	*a*	*b*	*c*	*d*	*e*	*f*	*f*	*g*	*f*	*g*	*a*
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

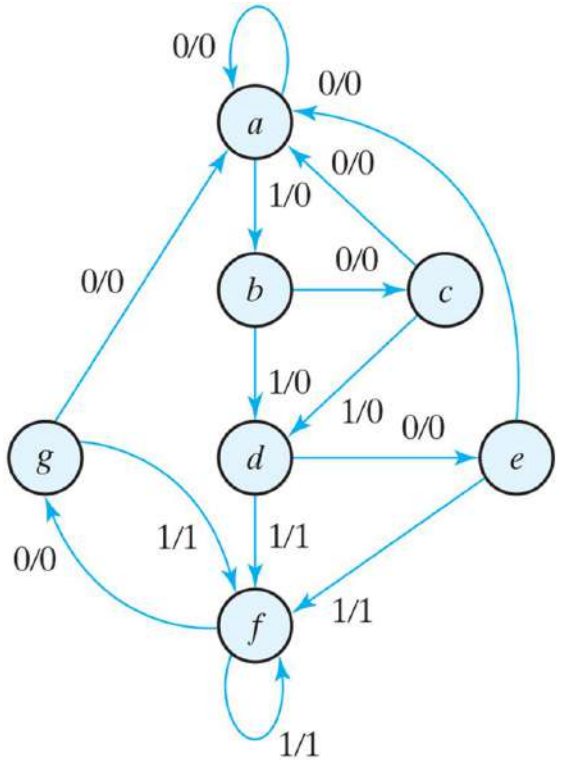


- We need to find a state diagram with less states that maintain the input-output relationship

State reduction example

- The first step is to get the state table
  - It is easier to work with table
- Here is the state table
  - Verify the table as an exercise

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1



State reduction example

- “**Two states** are said to be **equivalent if**, for **each** member of the set of **inputs**, they **give** exactly the **same output** and **send** the circuit either to the same state or to an **equivalent state**”
- If two states are equivalent, we can remove one of them without affecting the sequential circuit process.

States e, g both

have ***a*** as next state when **x =0**

Have ***f*** as next state when **x =1**

And produce the same output

States e, and g are equivalent one can be removed

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

State reduction example

- To remove a state **g**
  - The **row** with present state g is removed
  - In the row with states g as **next state** replace g with e

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	$\boxed{g}$ e	f	0	1
~~g~~	~~a~~	~~f~~	~~0~~	~~1~~

⇒

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

State reduction example

- Repeating the same steps again we can see that **f** and **d** are equivalent
- This gives us a table with one less state

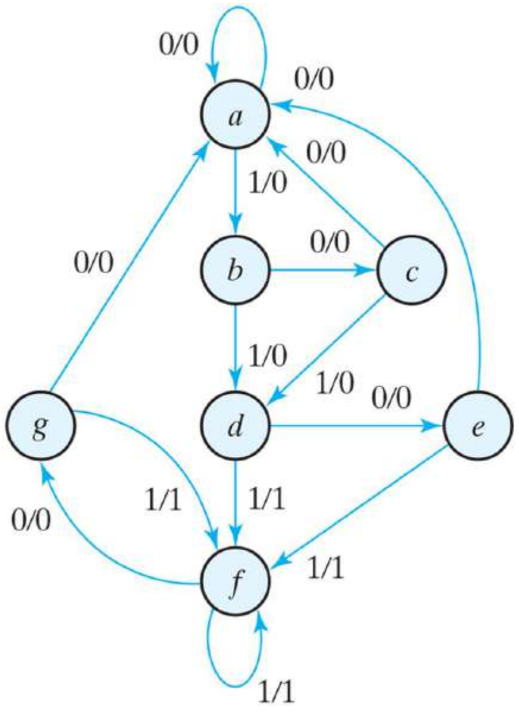
Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

$\Rightarrow$

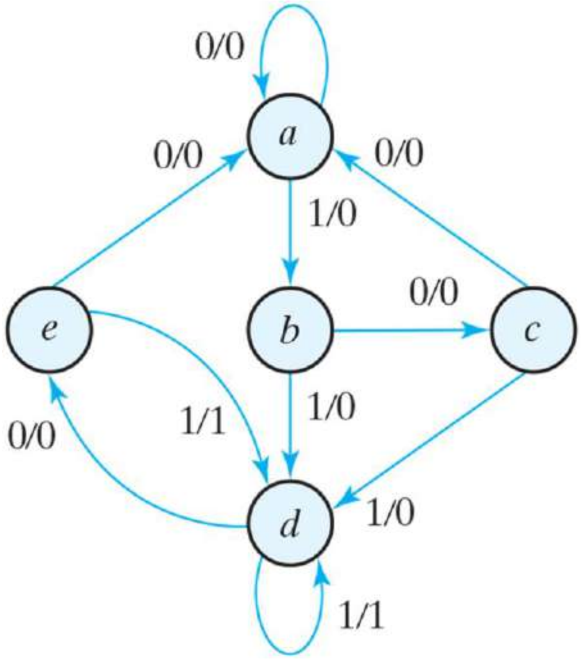
Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

State reduction example

- From the last state table, we can derive a new state diagram satisfying the same input-output relationships with less states



Original state diagram



reduced state diagram

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 13

Synchronous Sequential Logic

CONTENTS

- Registers
- Shift Registers
- Ripple Counters
- Synchronous Counters

Weekly Learning Outcomes

1. Understand the use, functionality, and modes of operation of registers, shift registers, and universal shift registers
2. Know how to properly create the effect of a gated clock.
3. Understand the structure and functionality of a serial adder circuit.
4. Understand the behavior of a (a) ripple counter, (b) synchronous counter

Required Reading

1. Chapter 6 (6.1 to 6.4)
(Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. Chapter 13
2. Chapter 14

(Digital Logic for Computing) (John Seiffertt Digital Logic for Computing)

Registers

Registers - Introduction

- A *register* is a group of flip-flops, each one of which shares a common clock and is capable of storing one bit of information. An *n*-bit register consists of a group of *n* flip-flops capable of storing *n* bits of binary information.
- In its broadest definition, a register consists of a group of flip-flops together with gates that affect their operation. The flip-flops hold the binary information, and the gates determine how the information is transferred into the register.

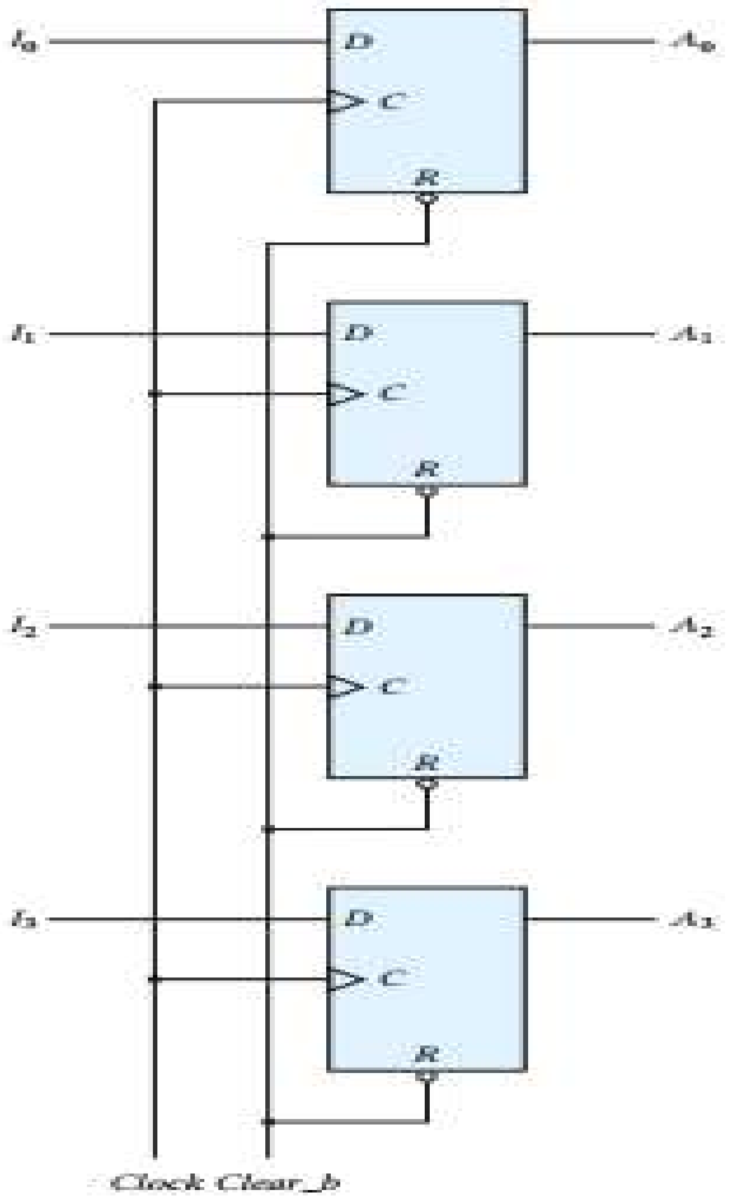
Counter

- A *counter* is essentially a register that goes through a predetermined sequence of binary states. The gates in the counter are connected in such a way as to produce the prescribed sequence of states.
- Although counters are a special type of register, it is common to differentiate them by giving them a different name.

Registers - Introduction

- The simplest register is one that consists of only flip-flops, without any gates.
- Such a register (As shown in next slide) constructed with four *D*-type flip-flops to form a four-bit data storage register.
- The common clock input triggers all flip-flops on the positive edge of each pulse, and the binary data available at the four inputs are transferred into the register.

Registers - Introduction



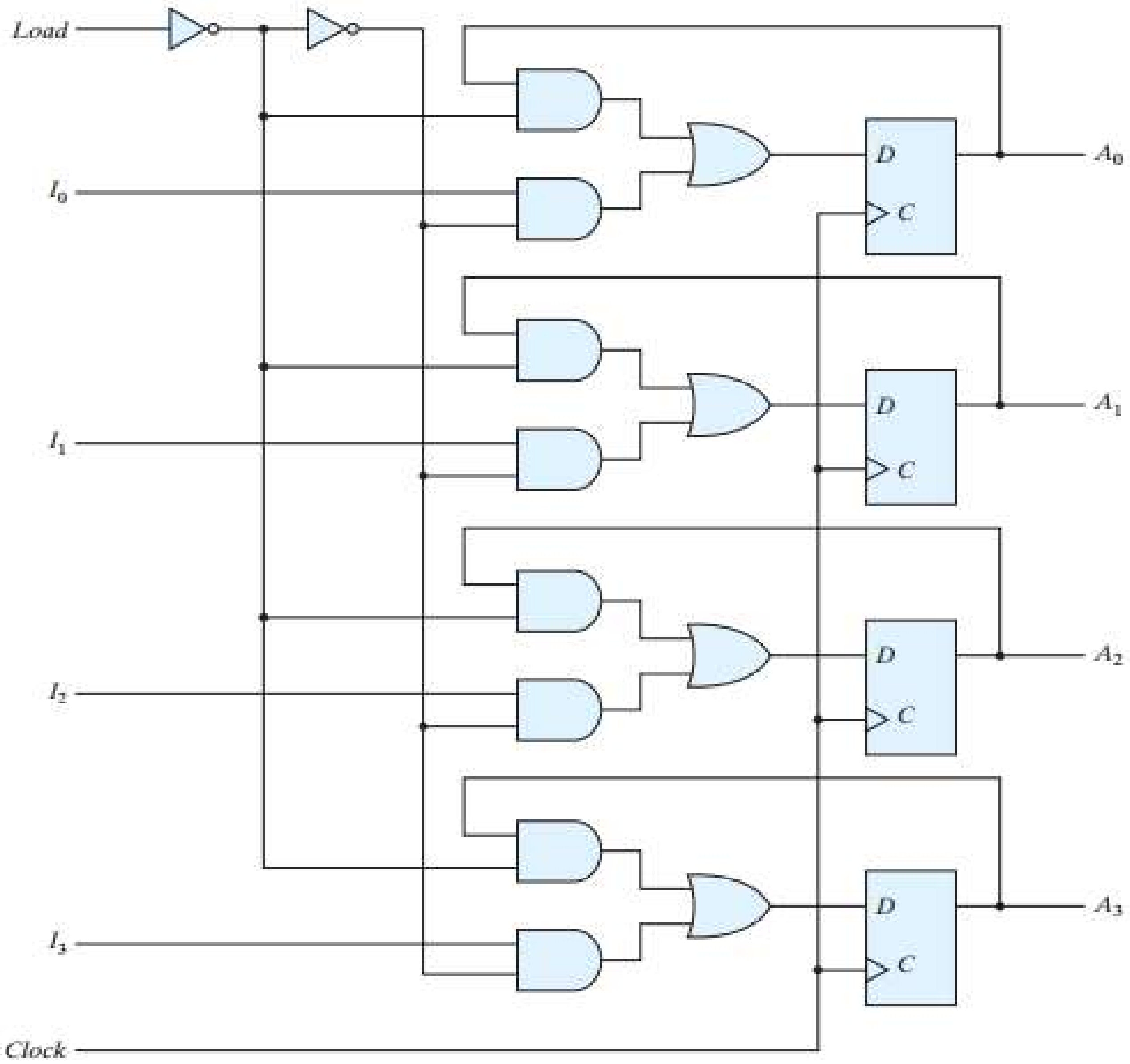
Registers - Introduction

- The value of (*I*3, *I*2, *I*1, *I*0) immediately before the clock edge determines the value of (*A*3,*A*2,*A*1,*A*0) after the clock edge. The four outputs can be sampled at any time to obtain the binary information stored in the register.
- The input *Clear_b* goes to the active-low *R* (reset) input of all four flip-flops. When this input goes to 0, all flip-flops are reset asynchronously. The *Clear_b* input is useful for clearing the register to all 0's prior to its clocked operation.
- The *R* inputs must be maintained at logic 1 (i.e., de-asserted) during normal clocked operation. Note that, depending on the flip-flop, either *Clear*, *Clear_b*, *reset*, or *reset_b* can be used to indicate the transfer of the register to an all 0's state.

Registers – With Parallel Load

- A four-bit data-storage register with a load control input that is directed through gates and into the *D* inputs of the flip-flops is shown in next slide.
- The additional gates implement a two-channel mux whose output drives the input to the register with either the data bus or the output of the register.

Registers – With Parallel Load



Registers – With Parallel Load

The load input to the register determines the action to be taken with each clock pulse.

- When the load input is 1, the data at the four external inputs are transferred into the register with the next positive edge of the clock.
- When the load input is 0, the outputs of the flip-flops are connected to their respective inputs.
- The feedback connection from output to input is necessary because a *D* flip-flop does not have a “no change” condition.

Registers – With Parallel Load

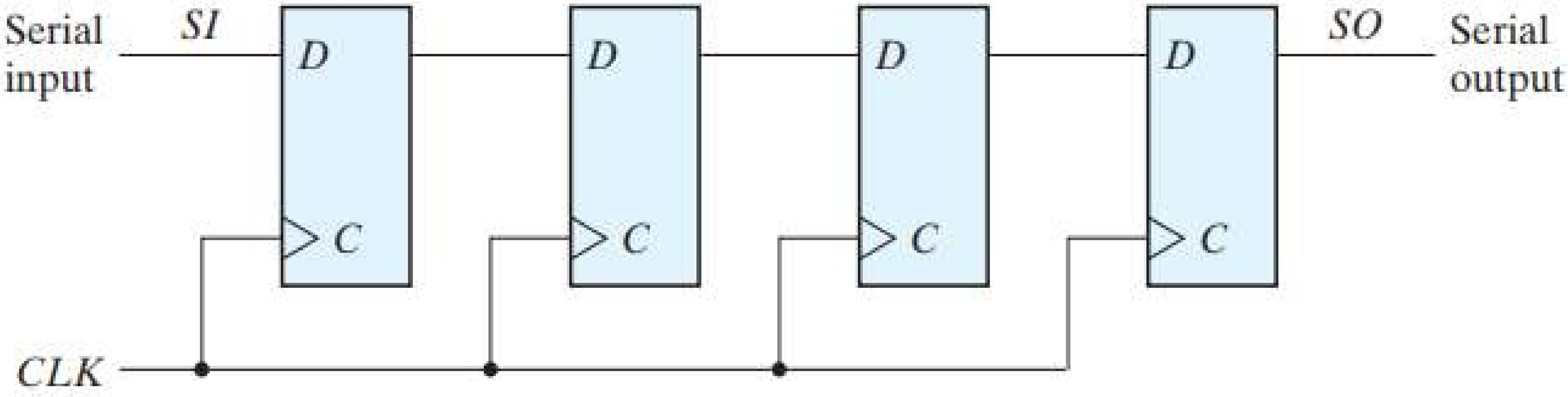
- The load input determines whether the next pulse will accept new information or leave the information in the register intact.
- The transfer of information from the data inputs or the outputs of the register is done simultaneously with all four bits in response to a clock edge.

Shift Registers

Shift Registers

- A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called a *shift register.*
- The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next.

Simplest Shift Registers



Simplest Shift Registers

- The output of a given flip-flop is connected to the *D* input of the flip-flop at its right.
- This shift register is unidirectional (left-to-right).
- Each clock pulse shifts the contents of the register one bit position to the right. The configuration does not support a left shift.
- The *serial input* determines what goes into the leftmost flip-flop during the shift. The *serial output* is taken from the output of the rightmost flip-flop.

Serial Transfer

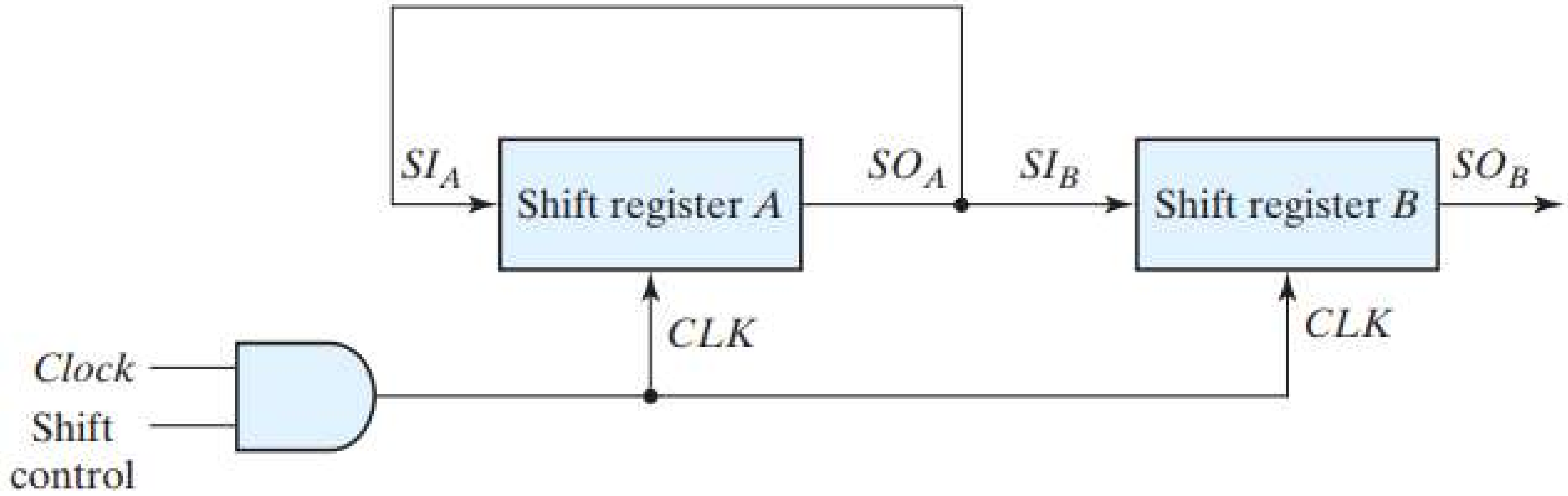
- The datapath of a digital system is said to operate in serial mode when information is transferred and manipulated one bit at a time.
- Information is transferred one bit at a time by shifting the bits out of the source register and into the destination register.
- This type of transfer is in contrast to parallel transfer, whereby all the bits of the register are transferred at the same time.

Shift Registers - Register A to Register B

- The serial output (*SO*) of register *A* is connected to the serial input (*SI*) of register *B.*
- To prevent the loss of information stored in the source register, the information in register *A* is made to circulate by connecting the serial output to its serial input.
- The initial content of register *B* is shifted out through its serial output and is lost unless it is transferred to a third shift register. The shift control input determines when and how many times the registers are shifted.

Serial Transfer - Register A to Register B

The serial transfer of information from register *A* to register *B* is done with shift registers, as shown below.



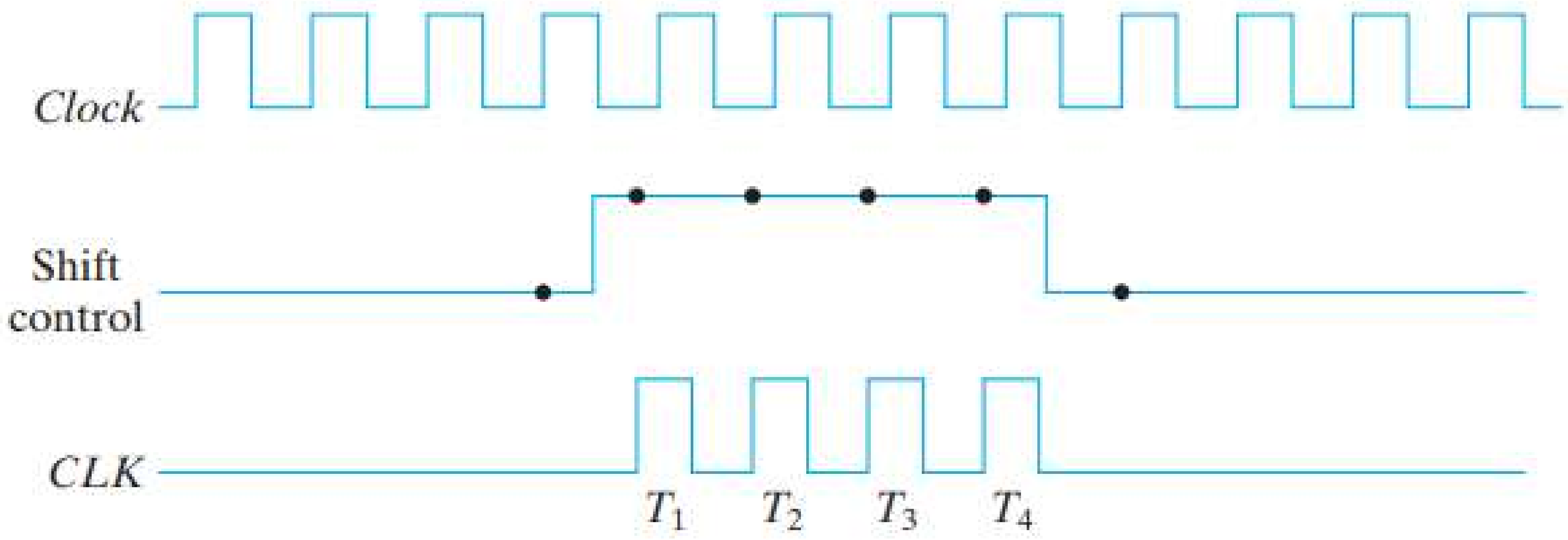
(a) Block diagram

Registers - Introduction

Suppose the shift registers in previous slide have four bits each. Then the control unit that supervises the transfer of data must be designed in such a way that it enables the shift registers, through the shift control signal, for a fixed time of four clock pulses in order to pass an entire word.

- This design is shown in the timing diagram in next slide.

Shift Registers – Timing Diagram



(b) Timing diagram

Shift Registers - Working

- The shift control signal is synchronized with the clock and changes value just after the negative edge of the clock.
- The next four clock pulses find the shift control signal in the active state, so the output of the AND gate connected to the *CLK* inputs produces four pulses: *T*1, *T*2, *T*3, and *T*4.
- Each rising edge of the pulse causes a shift in both registers. The fourth pulse changes the shift control to 0, and the shift registers are disabled.

Shift Registers - Example

Assume that the binary content of *A* before the shift is 1011 and that of *B* is 0010.

The serial transfer from *A* to *B* occurs in four steps, as shown in Table in the next slide.

Shift Registers - Example

Serial-Transfer Example

Timing Pulse	Shift Register *A*				Shift Register *B*			
Initial value	1	0	1	1	0	0	1	0
After T_1	1	1	0	1	1	0	0	1
After T_2	1	1	1	0	1	1	0	0
After T_3	0	1	1	1	0	1	1	0
After T_4	1	0	1	1	1	0	1	1

Shift Registers – Example Explanation

- With the first pulse, *T*1, the rightmost bit of *A* is shifted into the leftmost bit of *B* and is also circulated into the leftmost position of *A.*
- At the same time, all bits of *A* and *B* are shifted one position to the right.
- The previous serial output from *B* in the rightmost position is lost, and its value changes from 0 to 1.
- The next three pulses perform identical operations, shifting the bits of *A* into *B,* one at a time.

Shift Registers – Example Explanation

- After the fourth shift, the shift control goes to 0, and registers *A* and *B* both have the value 1011.
- Thus, the contents of *A* are copied into *B*, so that the contents of *A* remain unchanged i.e., the contents of *A* are restored to their original value.

Difference between Serial and Parallel mode

- In the parallel mode, information is available from all bits of a register and all bits can be transferred simultaneously during one clock pulse.
- In the serial mode, the registers have a single serial input and a single serial output. The information is transferred one bit at a time while the registers are shifted in the same direction.

Universal Shift Registers

The most general shift register has the following capabilities:

1. A *clear* control to clear the register to 0.

2. A *clock* input to synchronize the operations.

3. A *shift-right* control to enable the shift-right operation and the *serial input* and *output* lines associated with the shift right.

4. A *shift-left* control to enable the shift-left operation and the *serial input* and *output* lines associated with the shift left.

Universal Shift Registers

5. A *parallel-load* control to enable a parallel transfer and the *n* input lines associated with the parallel transfer.

6. *n* parallel output lines.

7. A control state that leaves the information in the register unchanged in response to the clock. Other shift registers may have only some of the preceding functions, with at least one shift op

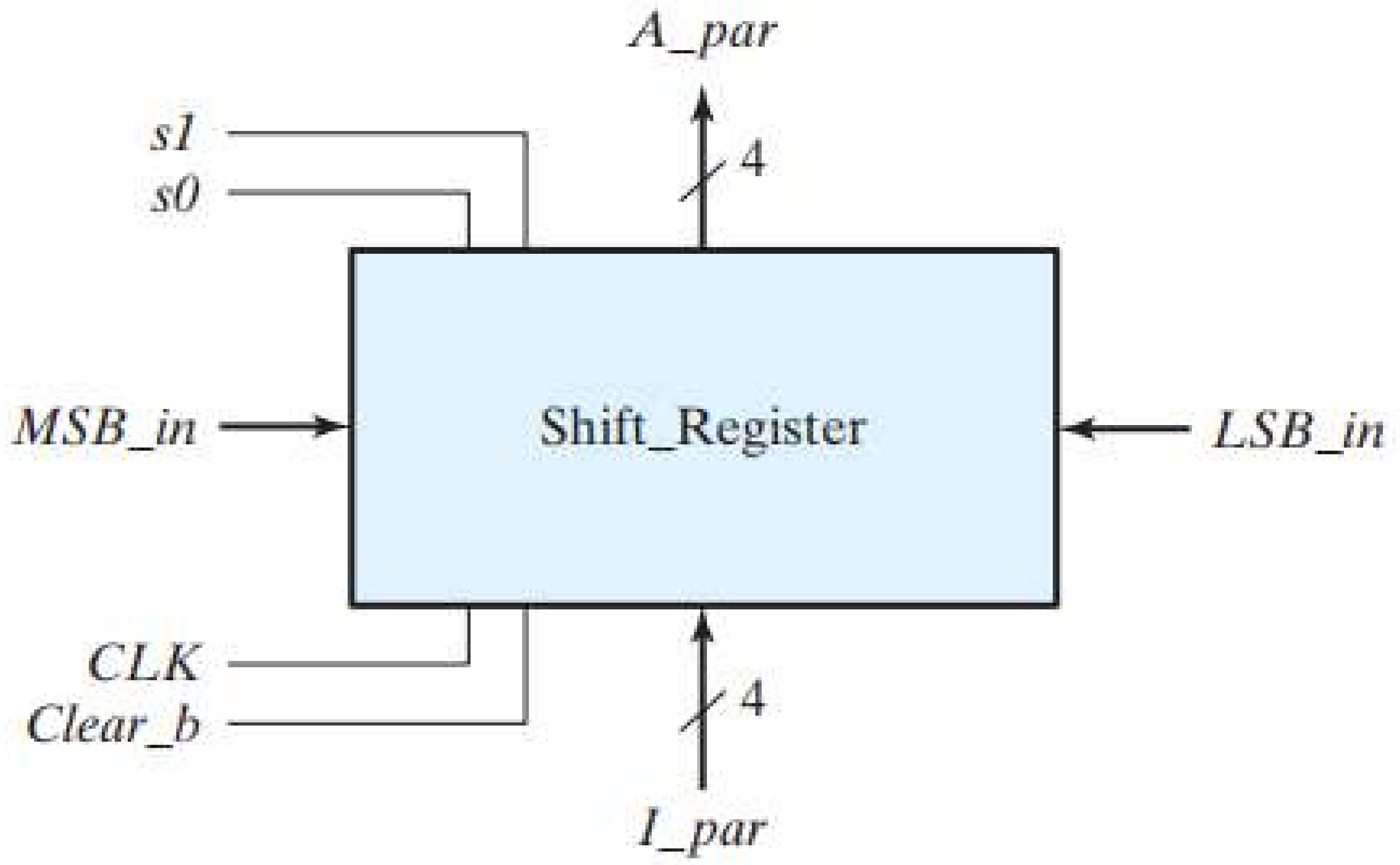
Universal Shift Registers

A register capable of shifting in one direction only is a ***unidirectional*** shift register.

One that can shift in both directions is a ***bidirectional*** shift register.

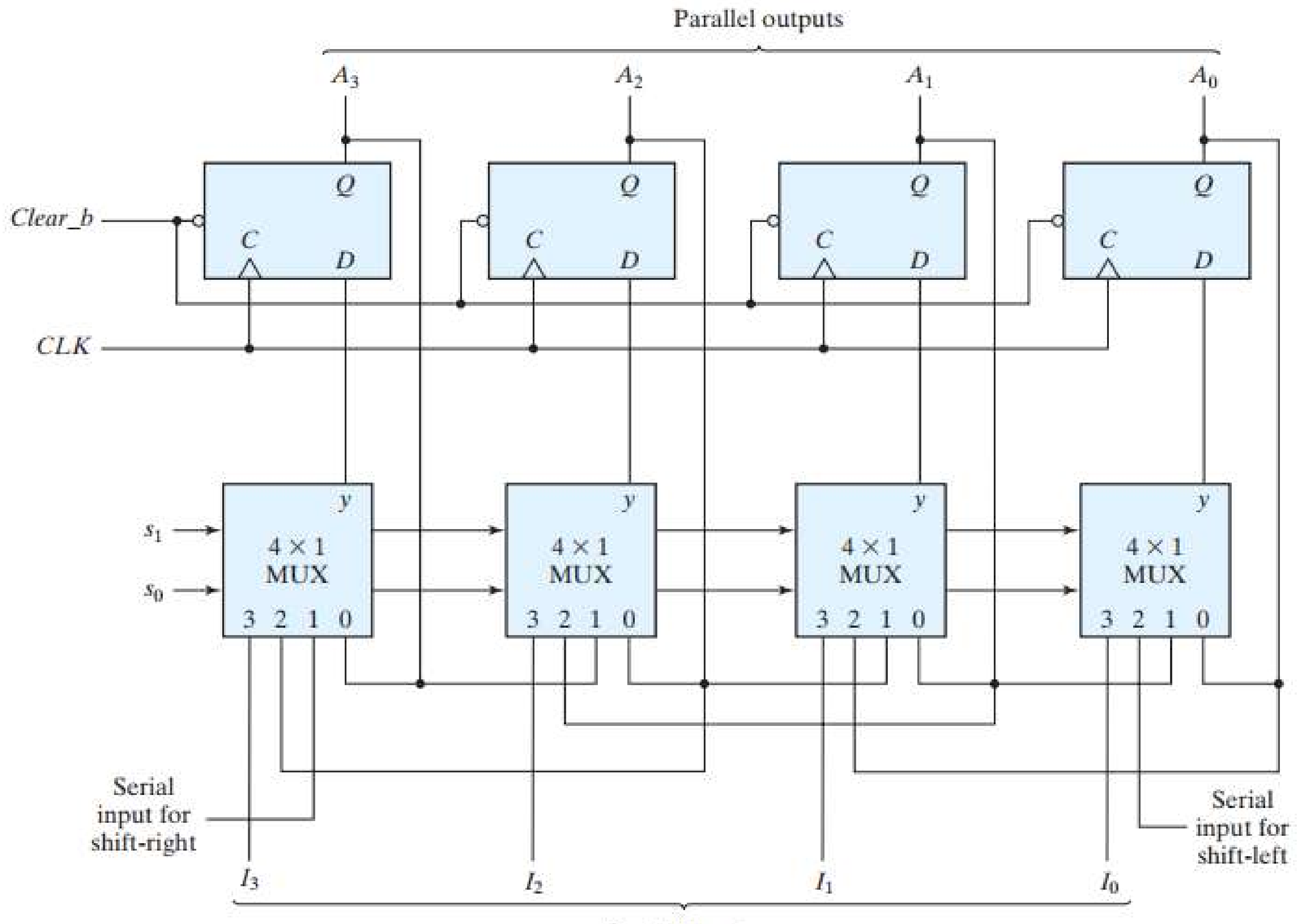
If the register has both shifts and parallel-load capabilities, it is referred to as a ***universal shift register****.*

Universal Shift Registers – 4 Bits



(a)

Universal Shift Registers – 4 Bits



(b)

Universal Shift Registers – 4 Bits

- The circuit consists of four *D* flip-flops and four multiplexers. The four multiplexers have two common selection inputs $s1$ and $s0$.
- Input 0 in each multiplexer is selected when $s1s0 = 00$, input 1 is selected when $s1s0 = 01$, and similarly for the other two inputs.
- The selection inputs control the mode of operation of the register according to the function entries in Table shown in next slide.

Universal Shift Registers – Function Table

Mode Control		
s_1	s_0	**Register Operation**
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

Universal Shift Registers - Working

- When s_1s_0 = 01, terminal 1 of the multiplexer inputs has a path to the *D* inputs of the flip-flops. This causes a shift-right operation, with the serial input transferred into flip-flop *A*3. When *s*1*s*0 = 10, a shift-left operation results, with the other serial input going into flip-flop *A*0.
- Finally, when s_1s_0 = 11, the binary information on the parallel input lines is transferred into the register simultaneously during the next clock edge.
- Note that data enters *MSB_in* for a shift-right operation and enters *LSB_in* for a shift-left operation. *Clear_b* is an active-low signal that clears all of the flip-flops.

Ripple Counters

Ripple Counter

- A register that goes through a prescribed sequence of states upon the application of input pulses is called a ***counter***.
- The sequence of states may follow the binary number sequence or any other sequence of states. A counter that follows the binary number sequence is called a ***binary counter***.
- An *n*-bit binary counter consists of *n* flip-flops and can count in binary from 0 through $2^n - 1$.

Ripple Counter

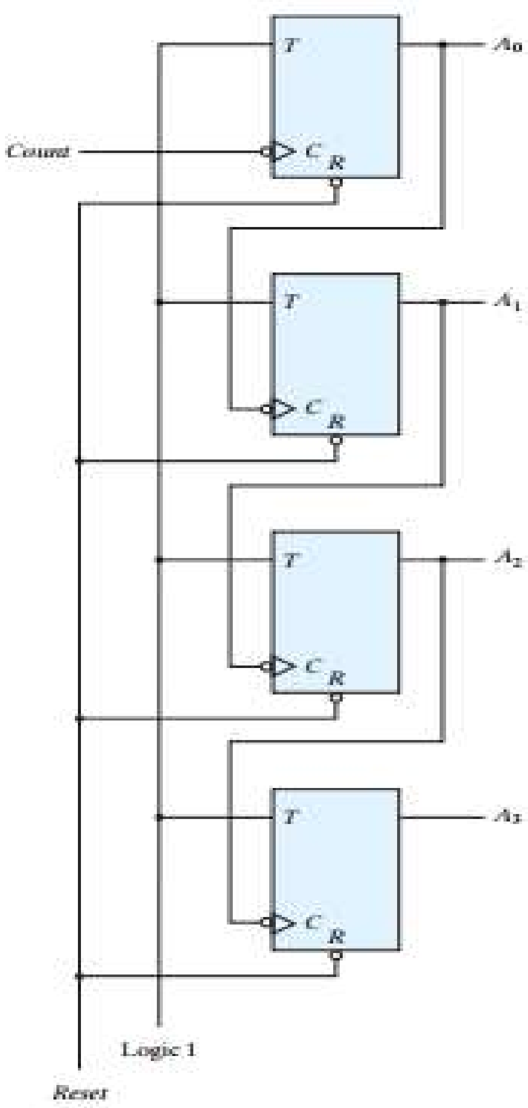
Counters are available in **two categories**: ripple counters and synchronous counters.

- In a **ripple counter**, a flip-flop output transition serves as a source for triggering other flip-flops. In other words, the *C* input of some or all flip-flops are triggered, not by the common clock pulses, but rather by the transition that occurs in other flip-flop outputs.
- In a **synchronous counter**, the *C* inputs of all flip-flops receive the common clock.

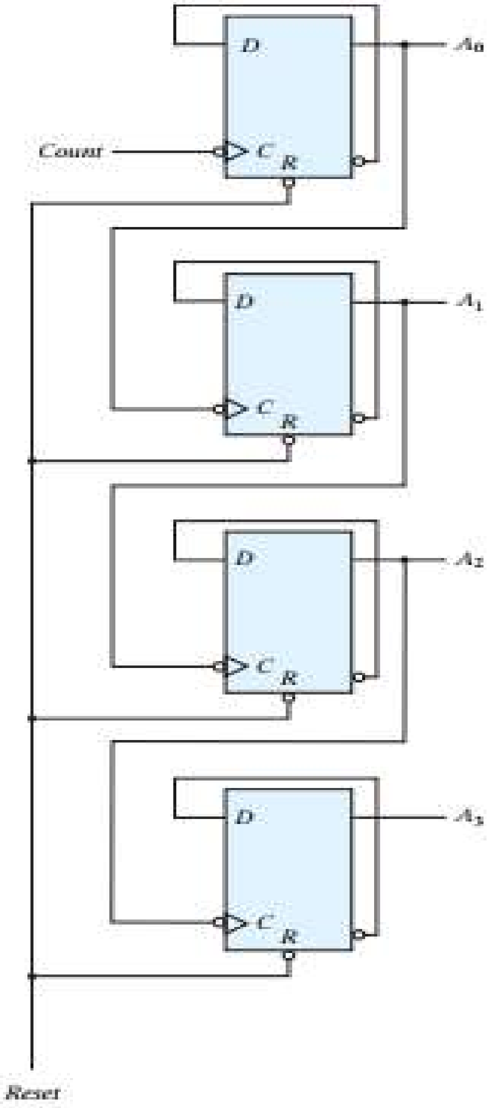
Binary Ripple Counter

- A binary ripple counter consists of a series connection of complementing flip-flops, with the output of each flip-flop connected to the *C* input of the next higher order flip-flop.
- The flip-flop holding the least significant bit receives the incoming count pulses.
- A complementing flip-flop can be obtained from a *JK* flip-flop with the *J* and *K* inputs tied together or from a *T* flip-flop.
- A third possibility is to use a *D* flip-flop with the complement output connected to the *D* input.

Four Bit Binary Ripple Counter – Logic Diagram



(a) With T flip-flops



(b) With D flip-flops

Four Bit Binary Ripple Counter – Operation

To understand the operation of the four-bit binary ripple counter, refer to the first nine binary numbers listed in Table below

Binary Count Sequence

A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0

Four Bit Binary Ripple Counter – Operation

- The count starts with binary 0 and increments by 1 with each count pulse input.
- After the count of 15, the counter goes back to 0 to repeat the count. The least significant bit, A_0, is complemented with each count pulse input.
- Every time that A_0 goes from 1 to 0, it complements A_1.
- Every time that A_1 goes from 1 to 0, it complements A_2.
- Every time that A_2 goes from 1 to 0, it complements A_3, and so on for any other higher order bits of a ripple counter.

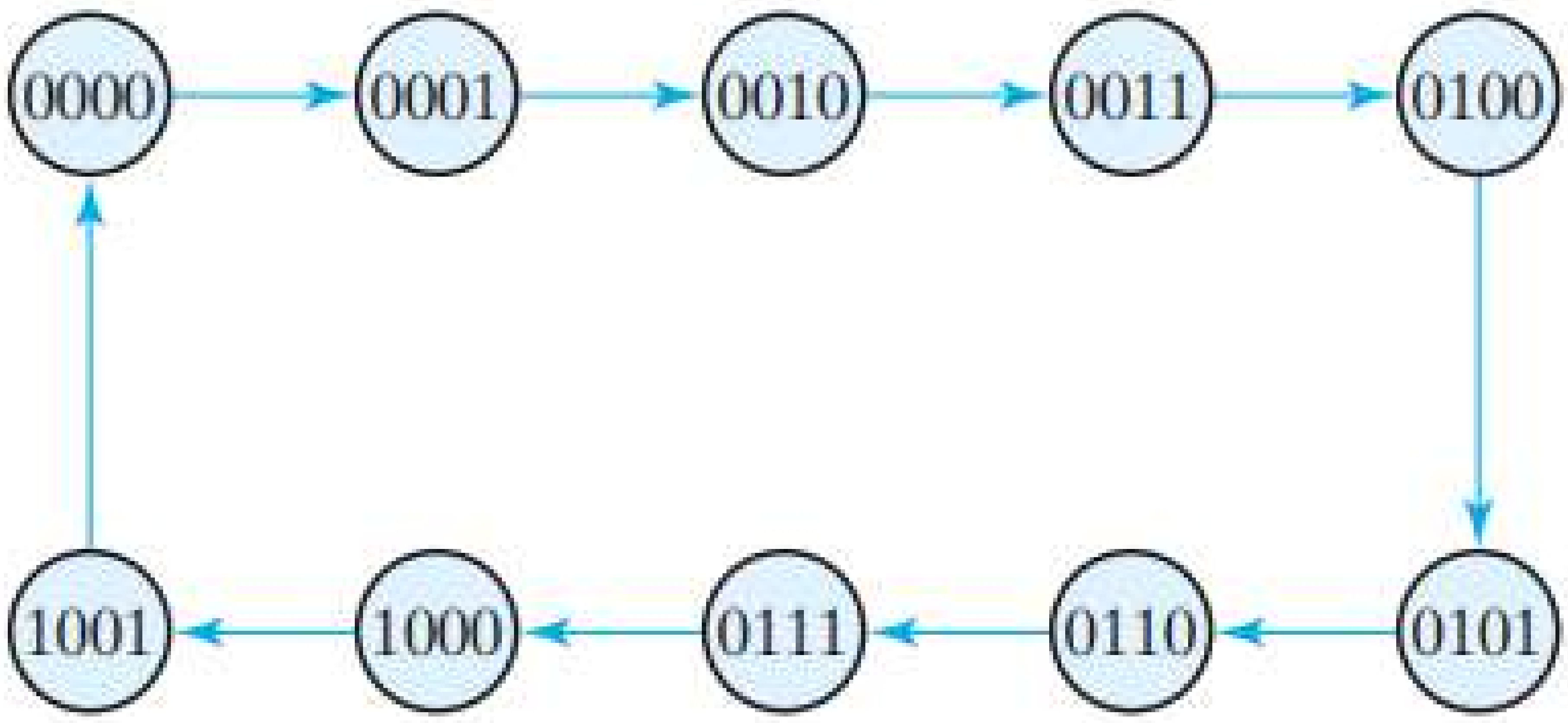
Binary Countdown Counter

- A binary counter with a reverse count is called a ***binary countdown counter***. In a countdown counter, the binary count is decremented by 1 with every input count pulse.
- The count of a four-bit countdown counter starts from binary 15 and continues to binary counts 14, 13, 12, . . . , 0 and then back to 15.

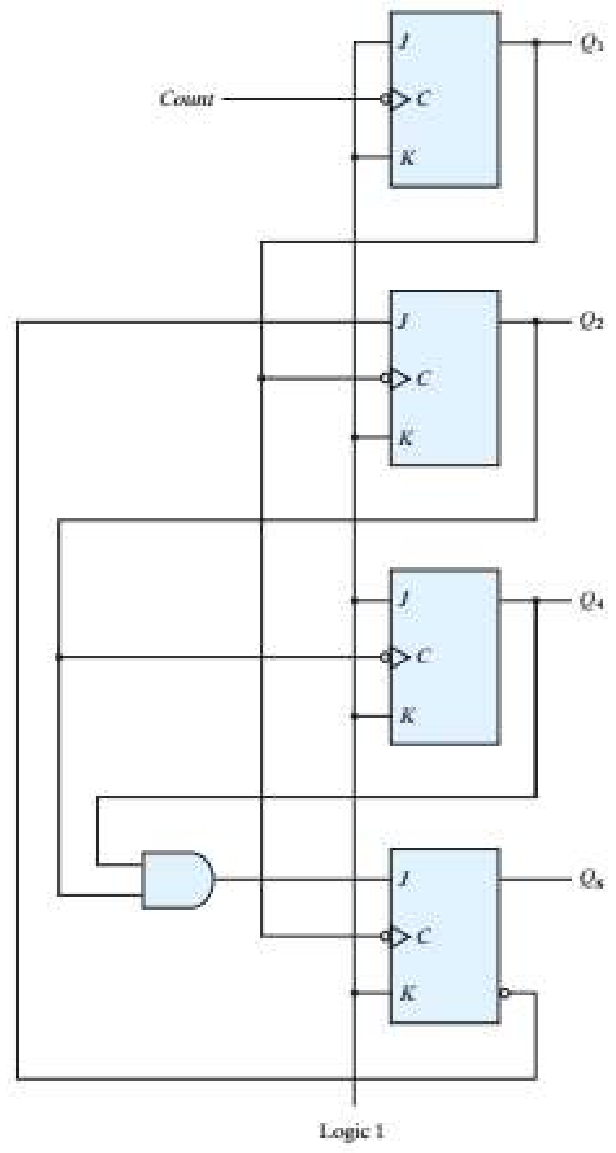
BCD Ripple Counter

- A decimal counter follows a sequence of 10 states and returns to 0 after the count of 9.
- Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits.
- If BCD is used, the sequence of states is as shown in the state diagram (in next slide)

BCD Ripple Counter – State Diagram



BCD Ripple Counter – Logic Diagram



BCD Ripple Counter

- Note that the output of $Q1$ is applied to the C inputs of both $Q2$ and $Q8$ and the output of $Q2$ is applied to the C input of $Q4$. The J and K inputs are connected either to a permanent 1 signal or to outputs of other flip-flops.

Synchronous Counters

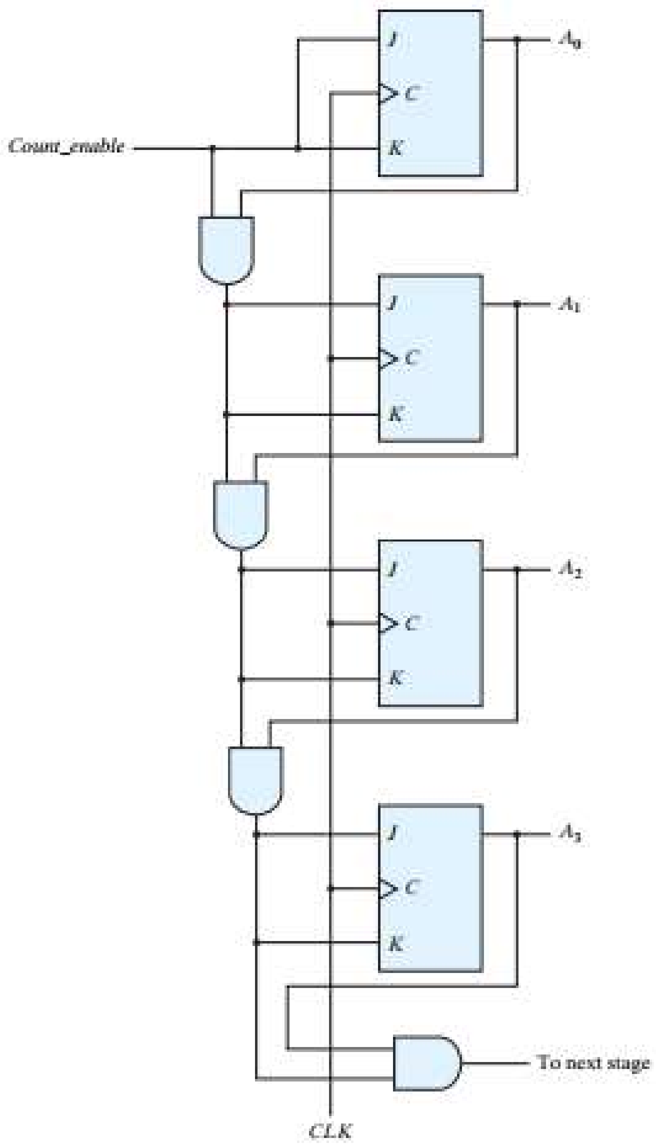
Synchronous Counter

- Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip-flops.
- A common clock triggers all flip-flops simultaneously, rather than one at a time in succession as in a ripple counter.

Synchronous Binary Counter

- In a synchronous binary counter, the flip-flop in the least significant position is complemented with every pulse.
- *A flip-flop in any other position is complemented when all the bits in the lower significant positions are equal to 1*.
- Synchronous binary counters have a regular pattern and can be constructed with complementing flip-flops and gates. The regular pattern can be seen from the four-bit counter depicted in next slide.

Synchronous Binary Counter



Synchronous Binary Counter

- The *C* inputs of all flip-flops are connected to a common clock. The counter is enabled by *Count_enable*.
- If the enable input is 0, all *J* and *K* inputs are equal to 0 and the clock does not change the state of the counter. The first stage, A_0, has its *J* and *K* equal to 1 if the counter is enabled.
- The other *J* and *K* inputs are equal to 1 if all previous least significant stages are equal to 1 and the count is enabled.
- The chain of AND gates generates the required logic for the *J* and *K* inputs in each stage. The counter can be extended to any number of stages, with each stage having an

  additional flip-flop and an AND gate that gives an output of 1 if all previous flip-flop outputs are 1.

BCD Synchronous Counter

- A BCD counter counts in binary-coded decimal from 0000 to 1001 and back to 0000. Because of the return to 0 after a count of 9, a BCD counter does not have a regular pattern, unlike a straight binary count.
- To derive the circuit of a BCD synchronous counter, it is necessary to go through a sequential circuit design procedure.

BCD Synchronous Counter – State Table

State Table for BCD Counter

Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

BCD Synchronous Counter – State Table

- The input conditions for the *T* flip-flops are obtained from the present- and next-state conditions.
- Also shown in the table is an output *y*, which is equal to 1 when the present state is 1001. In this way, *y* can enable the count of the next-higher significant decade while the same pulse switches the present decade from 1001 to 0000.

BCD Synchronous Counter – Map

- The flip-flop input equations can be simplified by means of maps. The unused states for minterms 10 to 15 are taken as don't-care terms. The simplified functions are

$$T_{Q1} = 1$$
$$T_{Q2} = Q_8'Q_1$$
$$T_{Q4} = Q_2Q_1$$
$$T_{Q8} = Q_8Q_1 + Q_4Q_2Q_1$$
$$y = Q_8Q_1$$

BCD Synchronous Counter

- The circuit can easily be drawn with four *T* flip-flops, five AND gates, and one OR gate. Synchronous BCD counters can be cascaded to form a counter for decimal numbers of any length.

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Computer Science Program
CS231
Digital Logic Design

CS231
Digital Logic Design

Week 14

Memory and Programmable Logic

Weekly Learning Outcomes

1. Learn the types and characteristics of memory (RAM, ROM)
2. Learn the memory read/write process
3. Learn how errors are detected and corrected while reading from memory
4. Learn the basics of how programable logic array (PLA, FPGA) work

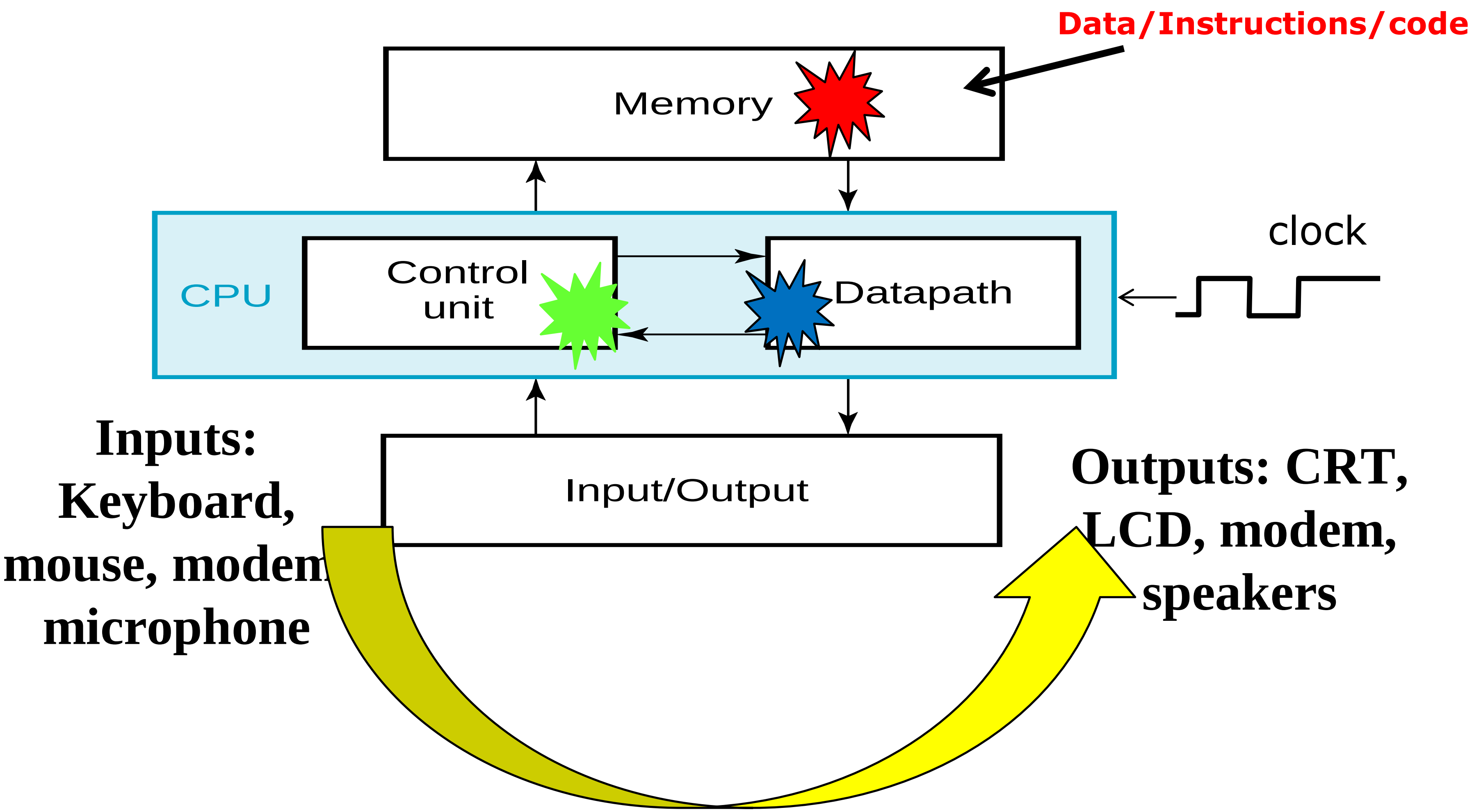
Required Reading

1. Chapter 7 (Sections 7.1- 7.8) (Digital Design with An Introduction to the Verilog HDL, VHDL and System Verilog)

Recommended Reading

1. https://www.geeksforgeeks.org/hamming-code-in-computer-network/

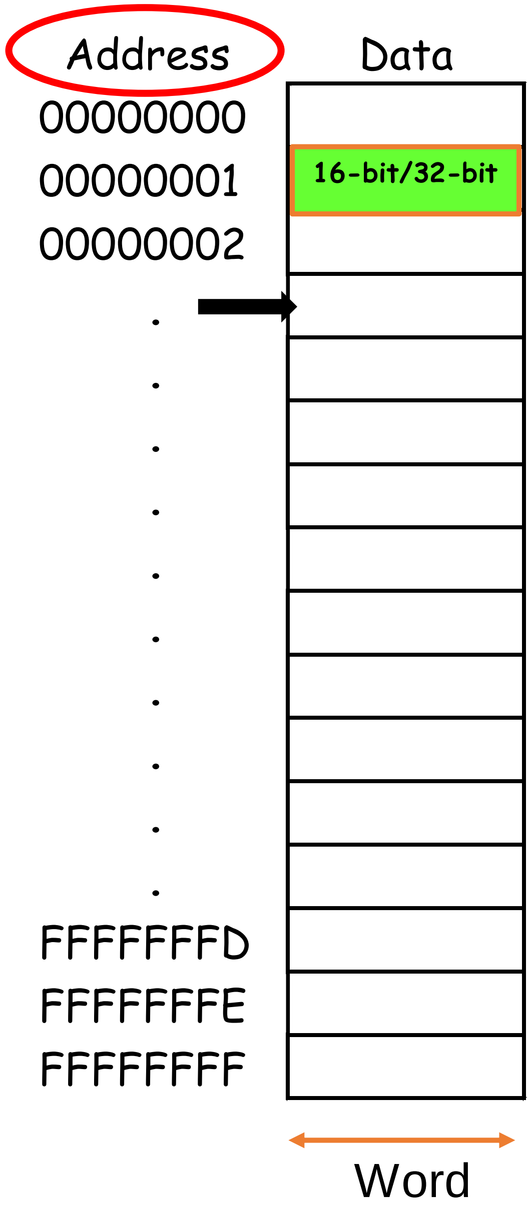
A Digital Computer System



Picture of Memory

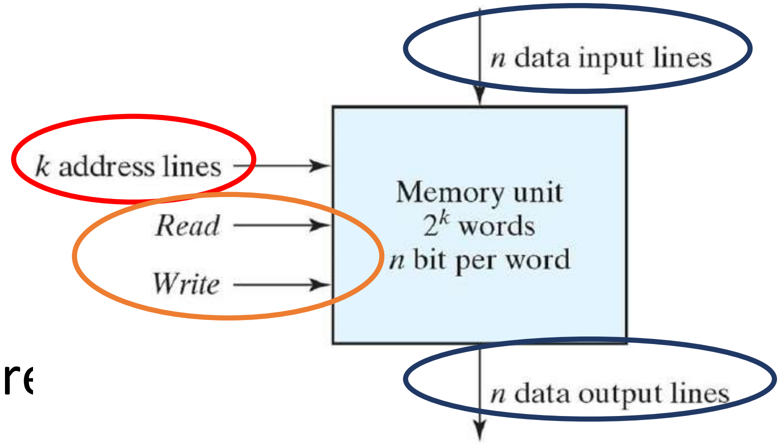
- You can think of memory as being one big **array of data**.
  - The address serves as an **array index**.
  - Each address refers to one word of data.

- You can **read** or **modify** the **data** at any given memory address, just like you can read or modify the contents of an array at any given index.



Memory Signal Types

- Memory signals fall into three groups
  - Address bus - selects one of memory locations
  - Data bus
    - Read: the selected location's stored data is put on the data bus
    - Write (RAM): The data on the data bus is store into selected locations
  - Control Signals - specifies what the memory is to do
    - Control signals are usually active low
    - Most common signals are:
      - CS: Chip Select; must be active to do anything
      - OE: Output Enable; active to enable data to appear on the bus
      - WR: Write; active to write data to memory
      - R: Read; active to read data from memory
      - Or a single R/W line



Properties of Memory

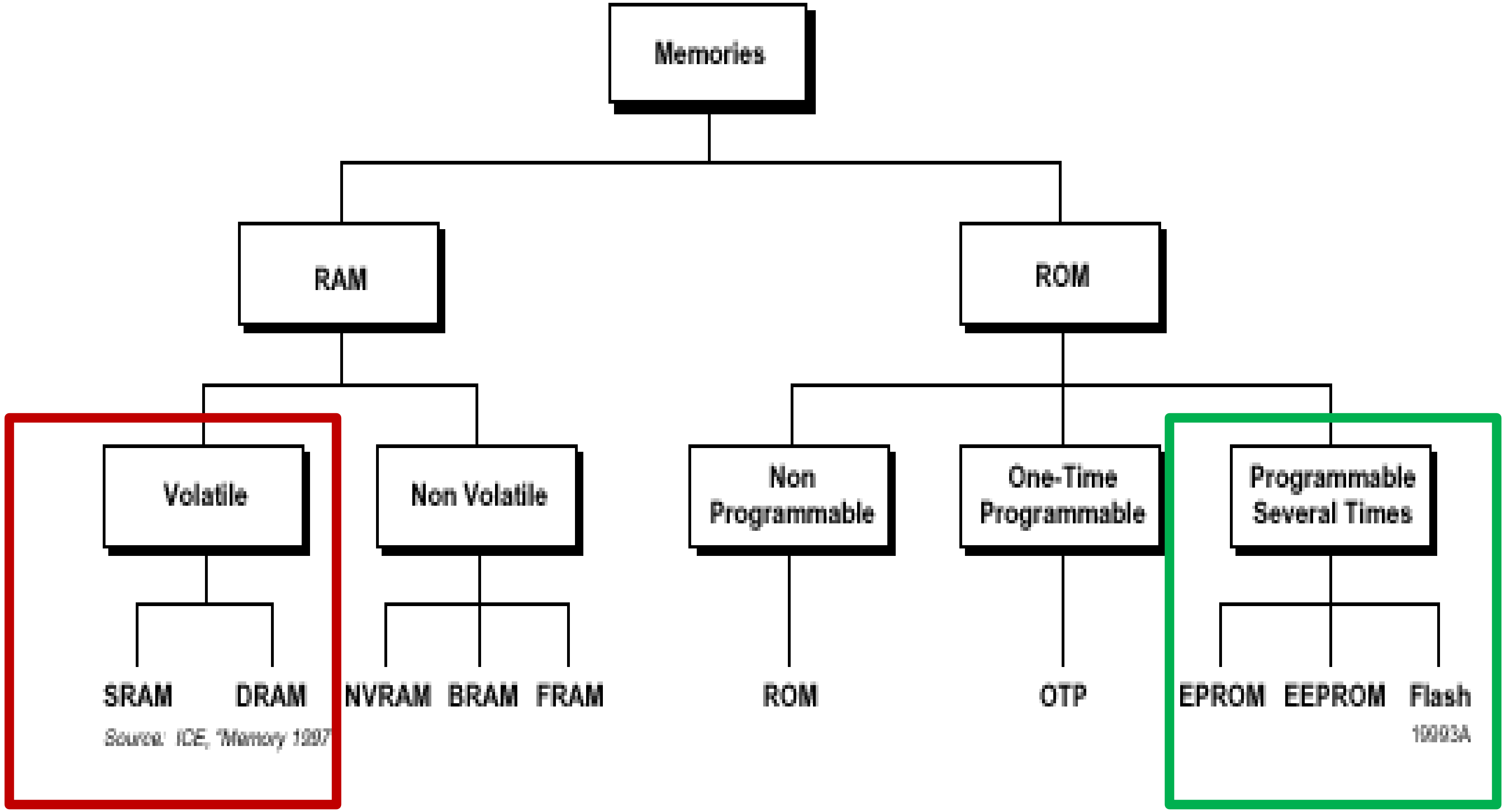
- Volatile: Memory contents ***disappears*** if power turned off, found in:
    - Typical computer systmes (laptops, desktops)
    - PDA, Smart Phone, iPADs, …
- Nonvolatile: Contents of memory ***remain*** even if power turned off, found in:
    - Smart Phones,
    - Hard Drives,
    - Memory Sticks

RAM vs. ROM

- ***Volatile Memory***
  - **RAM** (**R**andom **A**ccess **M**emory)
    - ***Static RAM*** usually used for Cache
    - ***Dynamic RAM*** used for Main Memory
- ***Non-Volatile Memory***
  - **ROM** (**R**ead **O**nly **M**emory),
    - EPROM
    - EEPROM
    - FLASH
    - Used to **store** permanent **programs** in a computer system (booting), **bit stream of FPGA**

Classification

Classification of Memory



Key Design Metrics

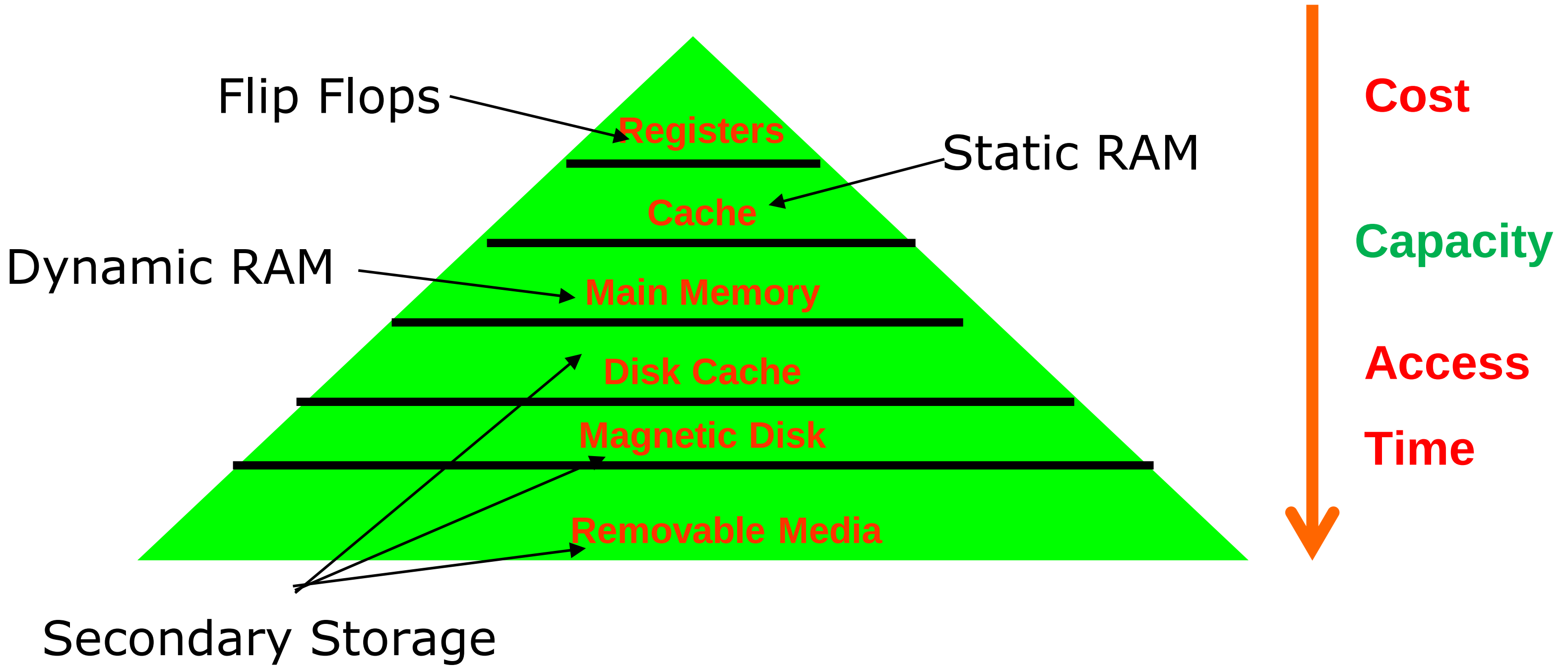
Read-Write Memory		Non-Volatile Read-Write Memory	Read-Only Memory (ROM)
Random Access	Non-Random Access	EPROM E^2PROM FLASH	Mask-Programmed
SRAM DRAM	FIFO LIFO		

Key Design Metrics:

1. Memory Density (number of bits/μm^2) and Size
2. Access Time (time to read or write) and Throughput
3. Power Dissipation

Memory Hierarchy

- The design constraints on a computer memory can be summed up by three questions (i) How Much (ii) How Fast (iii) How expensive.
- There is a ***tradeoff*** among the three key characteristics
- A variety of technologies are used to implement memory system
- Dilemma facing designer is clear → large capacity, fast, low cost!!
- Solution → Employ memory hierarchy



RAM vs. ROM

- RAM
  - ✓ Read/write
  - ✗ Volatile
  - ✓ Faster access time
  - ➢ Variants
    - ❑ SRAM
    - ❑ DRAM
  - Application
    - ❑ Variables
    - ❑ Dynamic memory allocation
    - ❑ Heaps, stacks

- ROM
  - Read only
  - Non-Volatile
  - Slower
  - Variants
    - PROM,EPROM
    - EEPROM, FLASH
  - Application
    - Programs
    - Constants
    - Codes, e.t.c

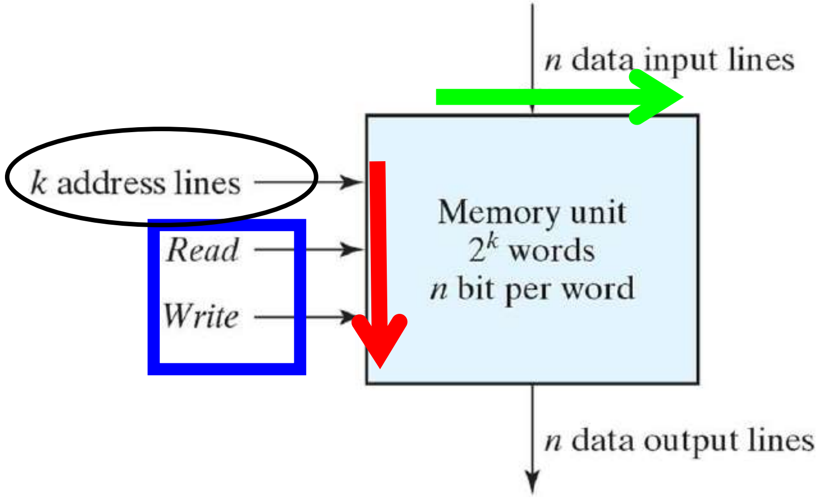
Random Access Memory

Random Access Memories

- So called because it **takes the same amount of time** to address/acess any location in the memory
- Types of RAM
  - Static RAM (SRAM), fast, expensive
  - Dynamic RAM (DRAM), slow, cheap
- How is memory accessed?
  - Address Lines, Data Lines
  - Control Signals (R/W, chip select, …)

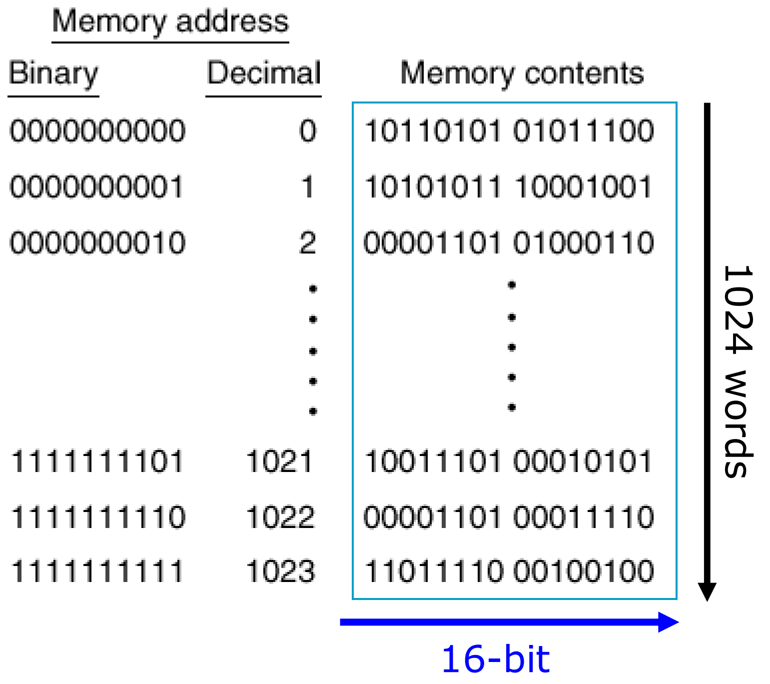
Simple View of RAM

- Of some word size n = 4,8,16,32,64 ....
  - n-bit data input lines
  - n-bit data output lines
- Some capacity 2^k
  - k bits of address line
- Has a read line
- Has a write line



1K x 16 memory

- Variety of sizes
  - From 1-bit wide
- Issue is no. of pins
- Memory size specified in bytes
  - 1K x 16 bit → 2KB memory
- 10 address lines and 16 data lines



Chip Select and R/W Lines

- **R/W** Lines enable reading/writing
- Usually, a **chip select** line is used.
- Why?
  - To enable RAM chip to be accessed.
  - Tri state capability

Control Inputs to a Memory Chip

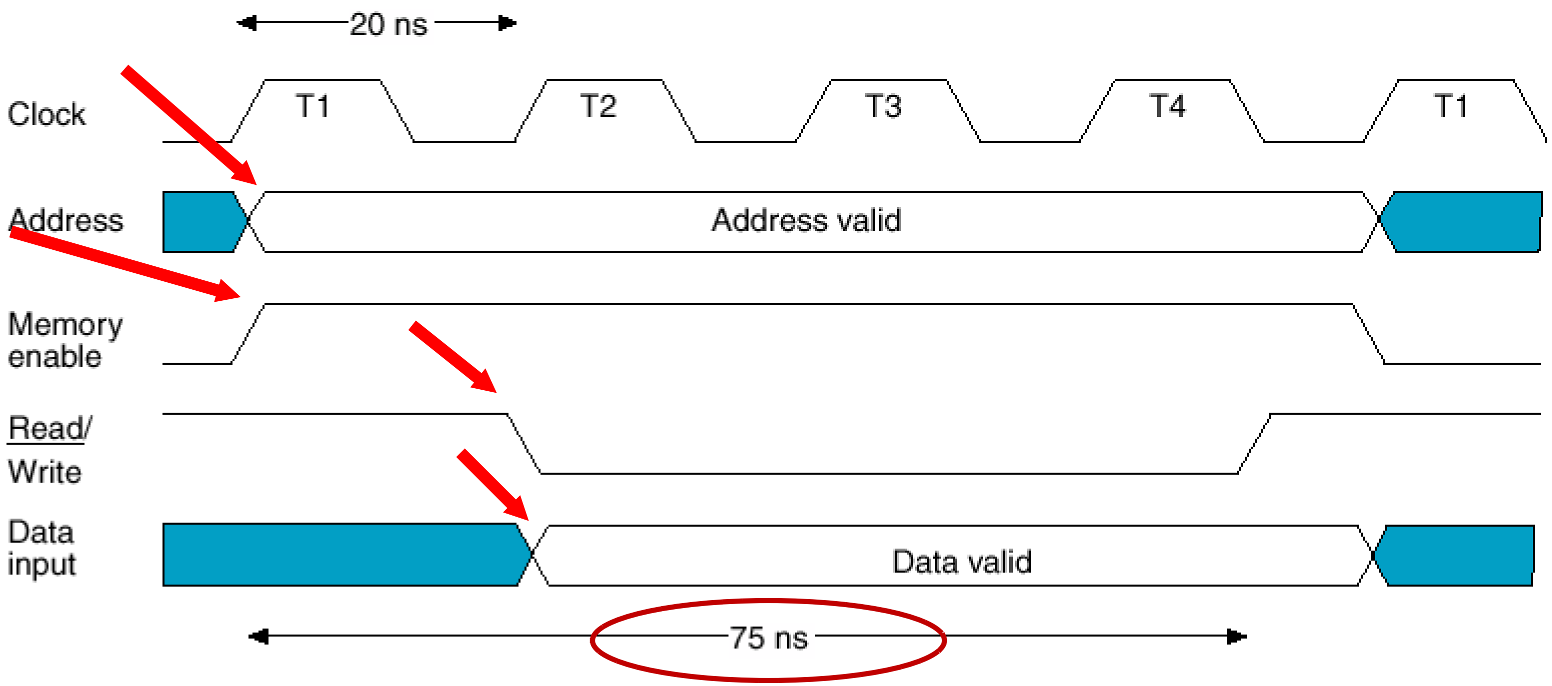
Chip select CS	Read/$\overline{\text{Write}}$ R/$\overline{\text{W}}$	Memory operation
0	×	None
1	0	Write to selected word
1	1	Read from selected word

Memory: Writing

- Sequence of steps
  - **Setup address** lines
  - **Setup data** lines
  - **Activate write** line (maybe a positive edge)
- The **write cycle** time is the **maximum time** from the application of the address to the completion of all internal memory operations required to store a word.

Writing: Timing Waveforms

- CPU operates at 50 MHz (20 ns)
- 4 clock cycles to perform a write



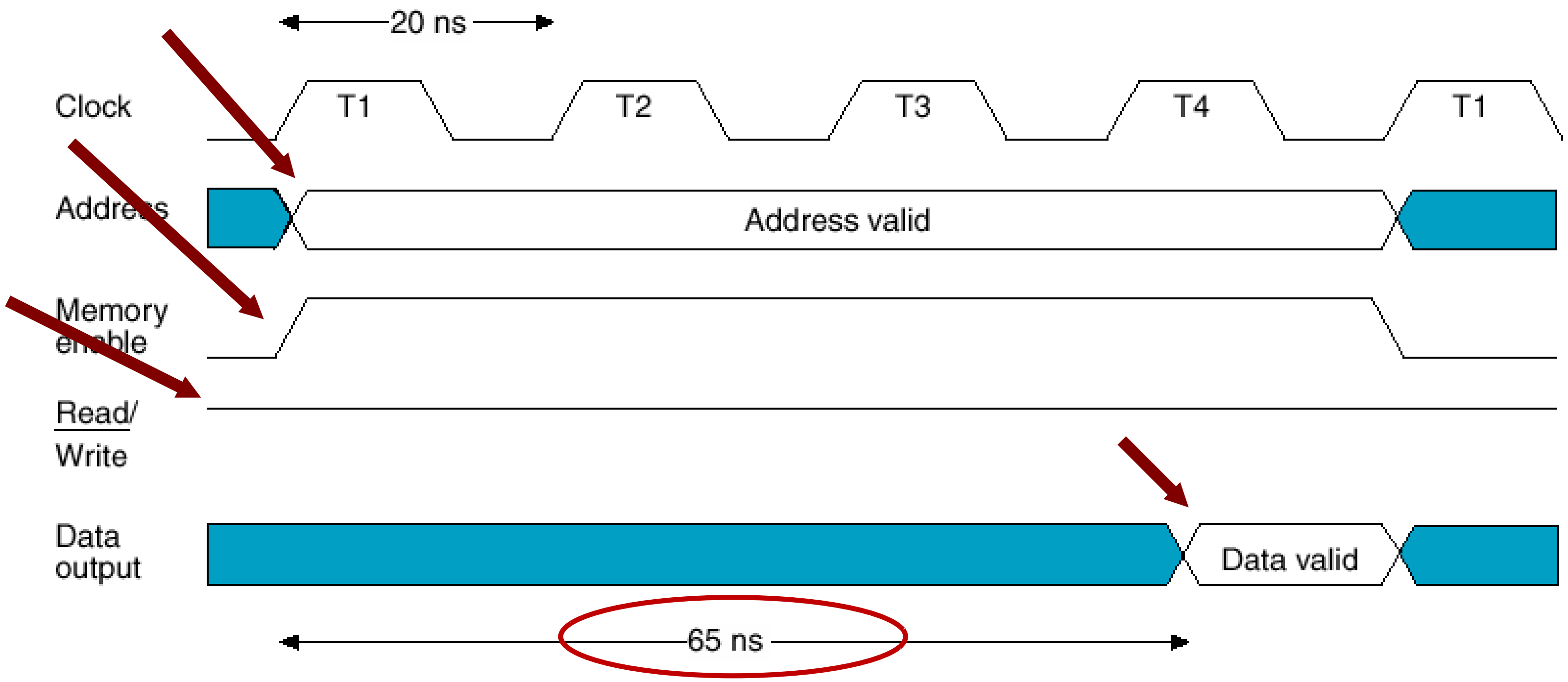
Memory Reading

- Steps:
  - **Setup address** lines
  - **Activate read** line
  - **Data available after** specified amount of **time**

- **Read cycle** usually is **shorter** than write cycle.

Memory Waveform: Reading

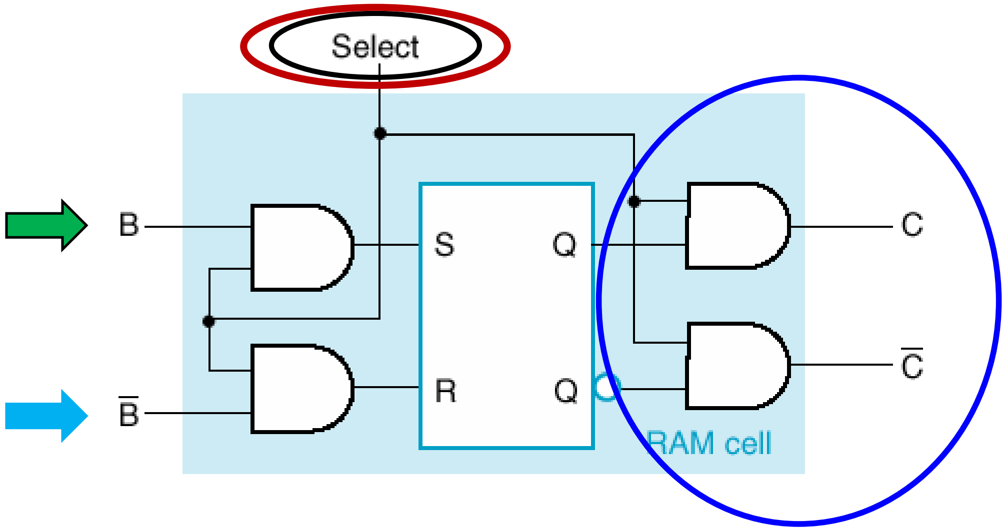
- CPU operates at 50 MHz (20 ns)
- 65 ns required for a read cycle



Static RAM

Simplify Modeling using Latch

- Static RAM storage can be modeled by an SR latch.
- Control logic
- One memory cell per bit



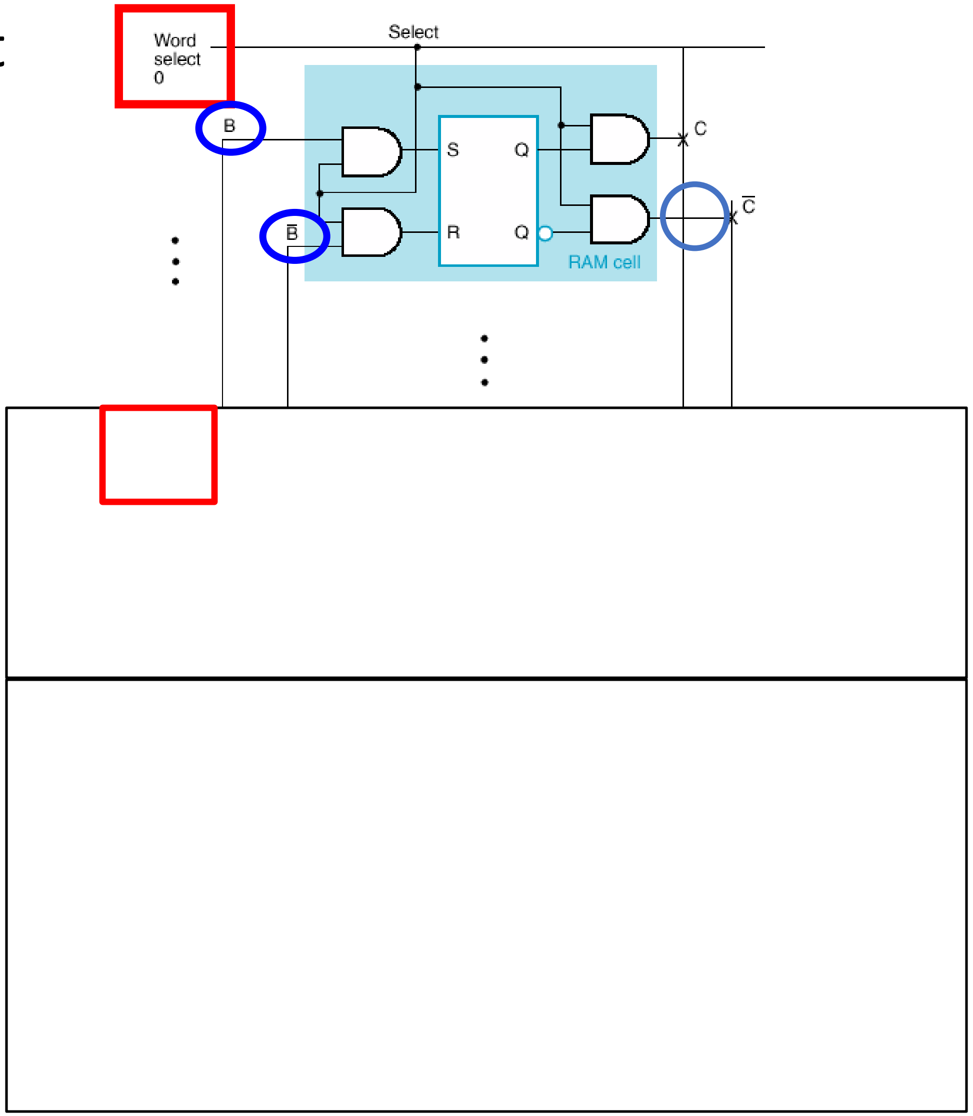
If select = 0, the stored content is held.

If select = 1, the stored content is determined by values on B and B'

The outputs are gated by the select line also.

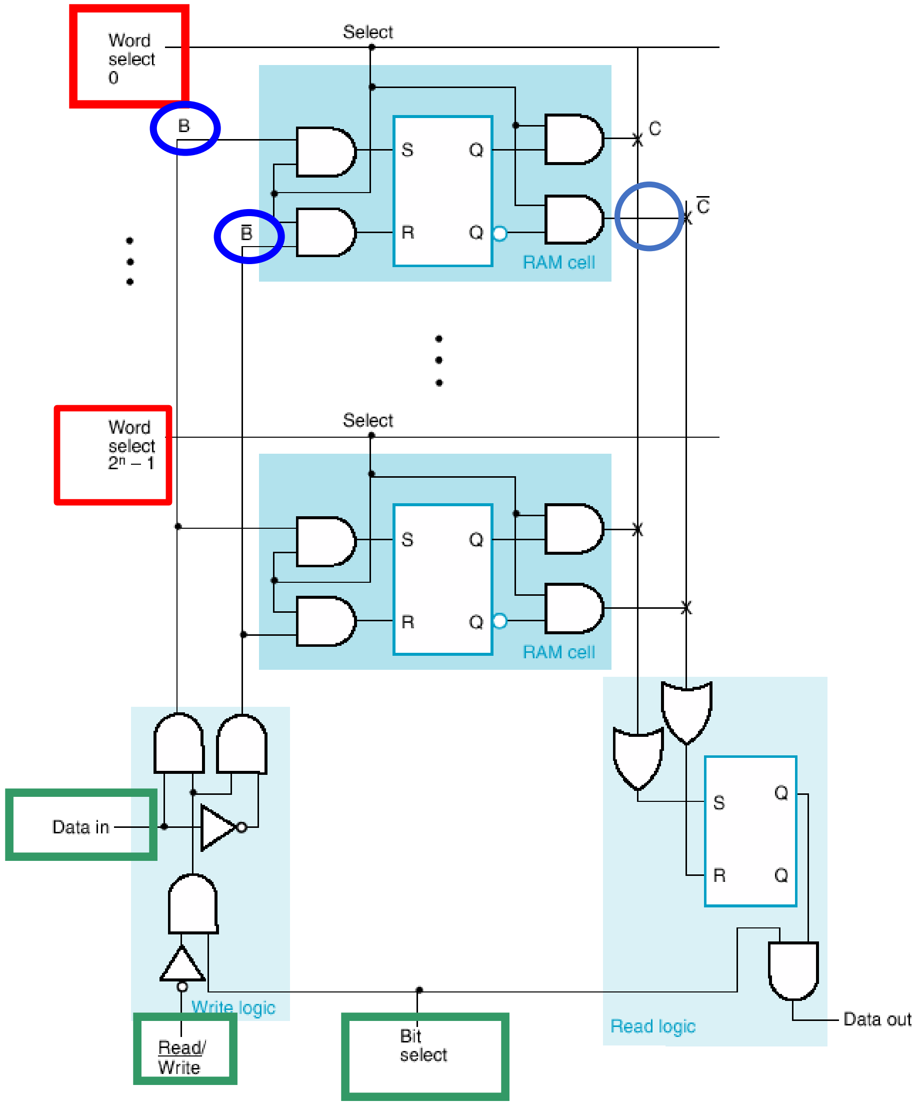
Bit Slice

- Cells connected to form 1-bit position (Slice)
- Word Select: selects a latch for read/write operations
- B (and B') set by R/W, Data In and BitSelect
- When R/W = 0 and BitSelect = 1,
- Then if Data in = 1
- The latch will be set
- A "1" is written



Bit Slice

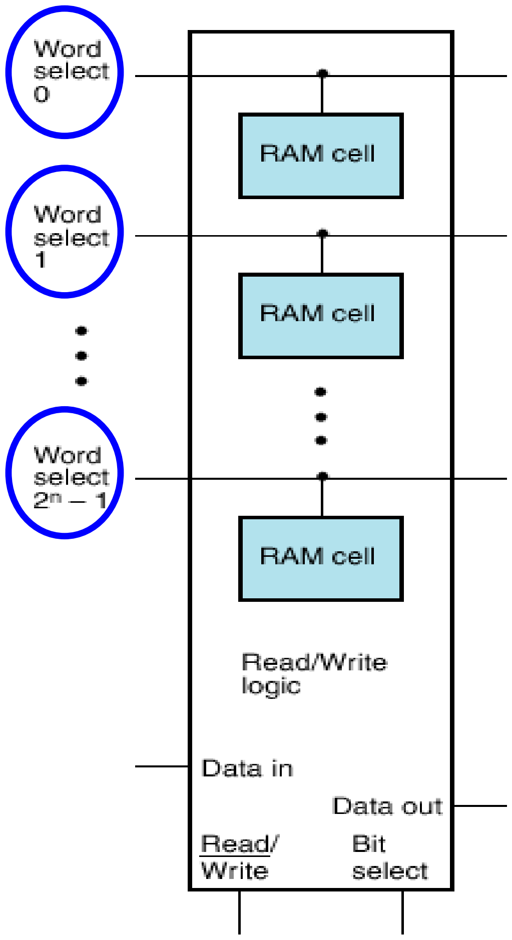
- Cells connected to form 1-bit position (Slice)
- Word Select: selects a latch for read/write operations
- B (and B') set by R/W, Data In and BitSelect
- When R/W = 0 and BitSelect = 1,
- Then if Data in = 1
- The latch will be set
- A "1" is written



Bit Slice can Become Module

- Basically, a bit slice is a one-dimensional array of memory
- What type of hardware should we use to access one row at a time?

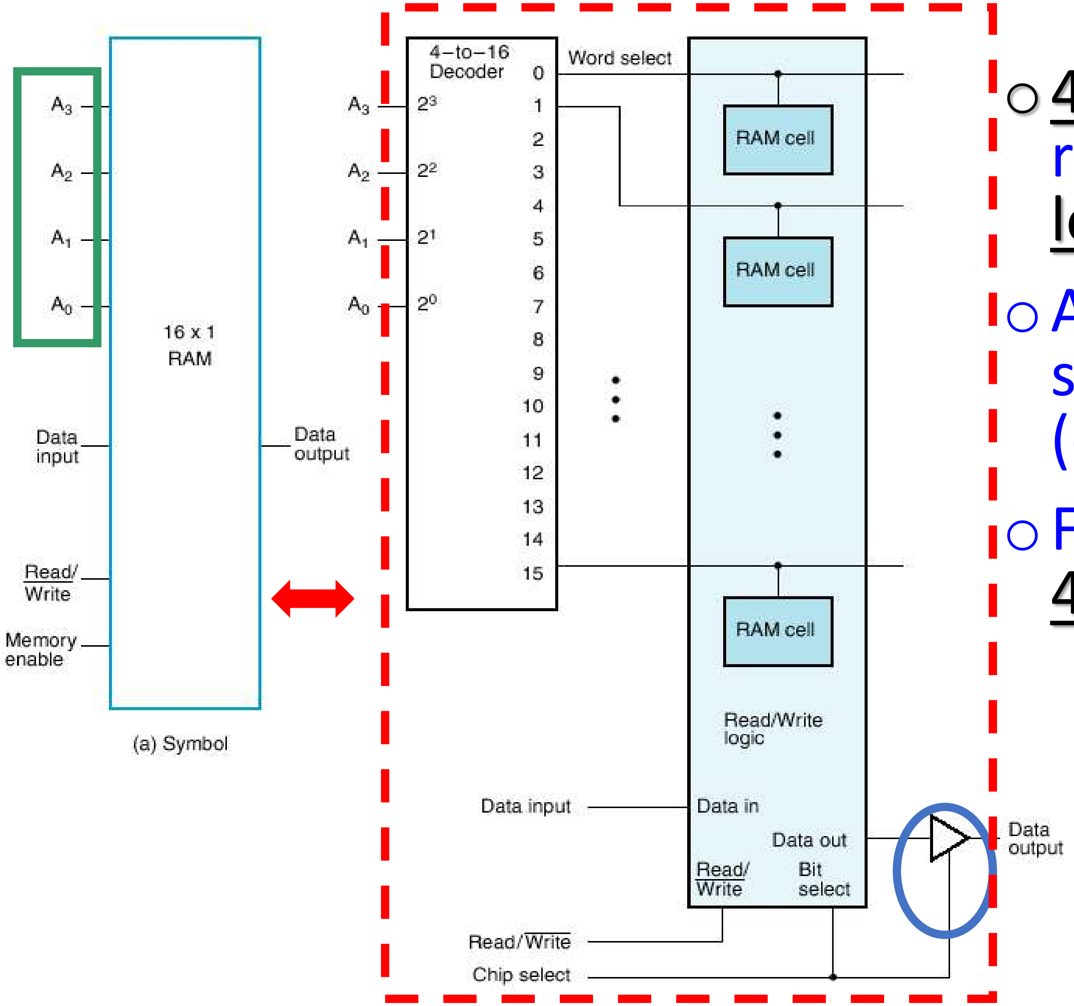
DECODERS!!!



(b) Symbol

16 X 1 RAM

Remember: A (4-to-16 line decoder) → equivalent to 16 4-input AND Gates



- 4 address lines are required to access 16 locations.
- A Decoder is added to select the different words (each 1 bit wide).
- For 16 words we need a 4-to-16 line Decoder

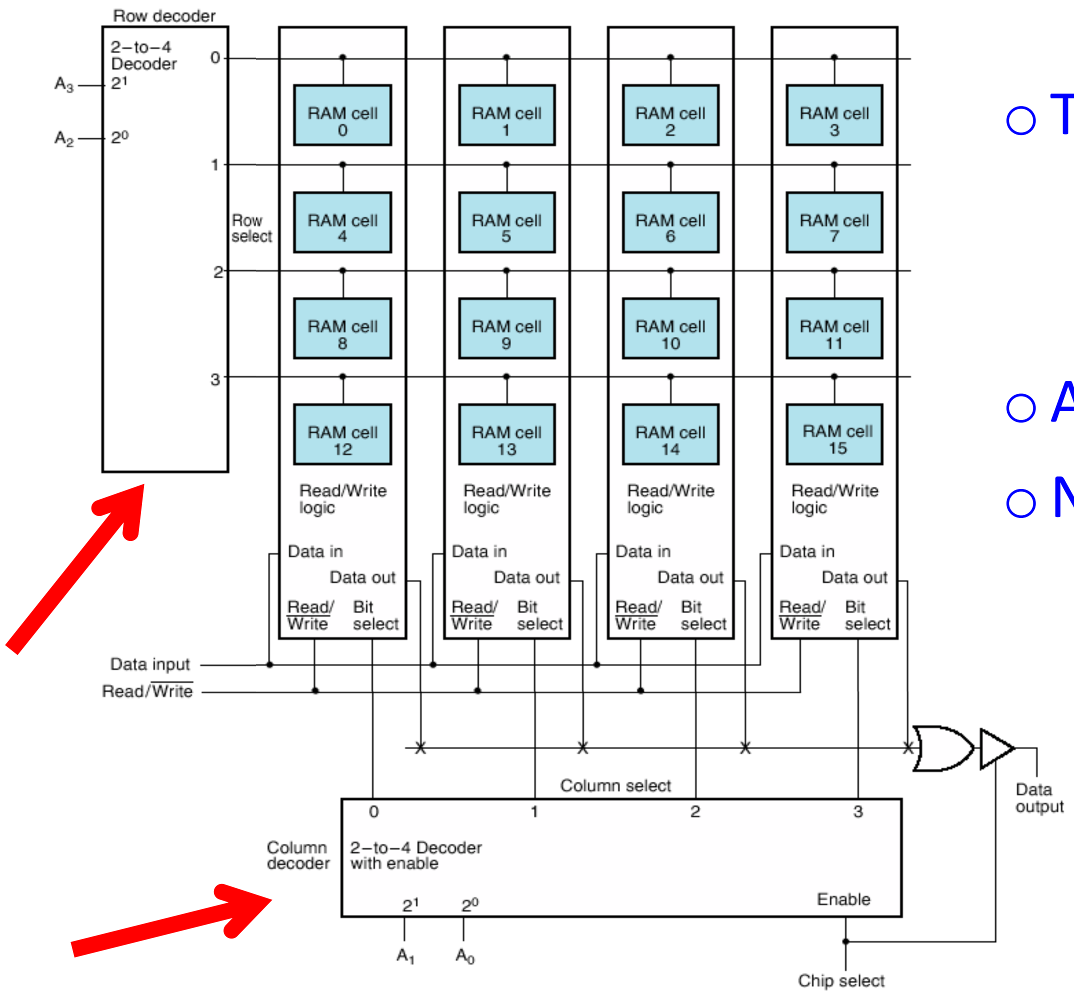
Row/Column

- Practical memories contains thousands of words!!
- If RAM gets large, there is a huge decoder
    - A 1-bit slice with 2048 locations needs 11-to-2048 Dec.
- Also run into chip layout issues
- How can we change the structure of Memory to solve this problem?

Rearrange the memory into “2D” i.e., matrix layout

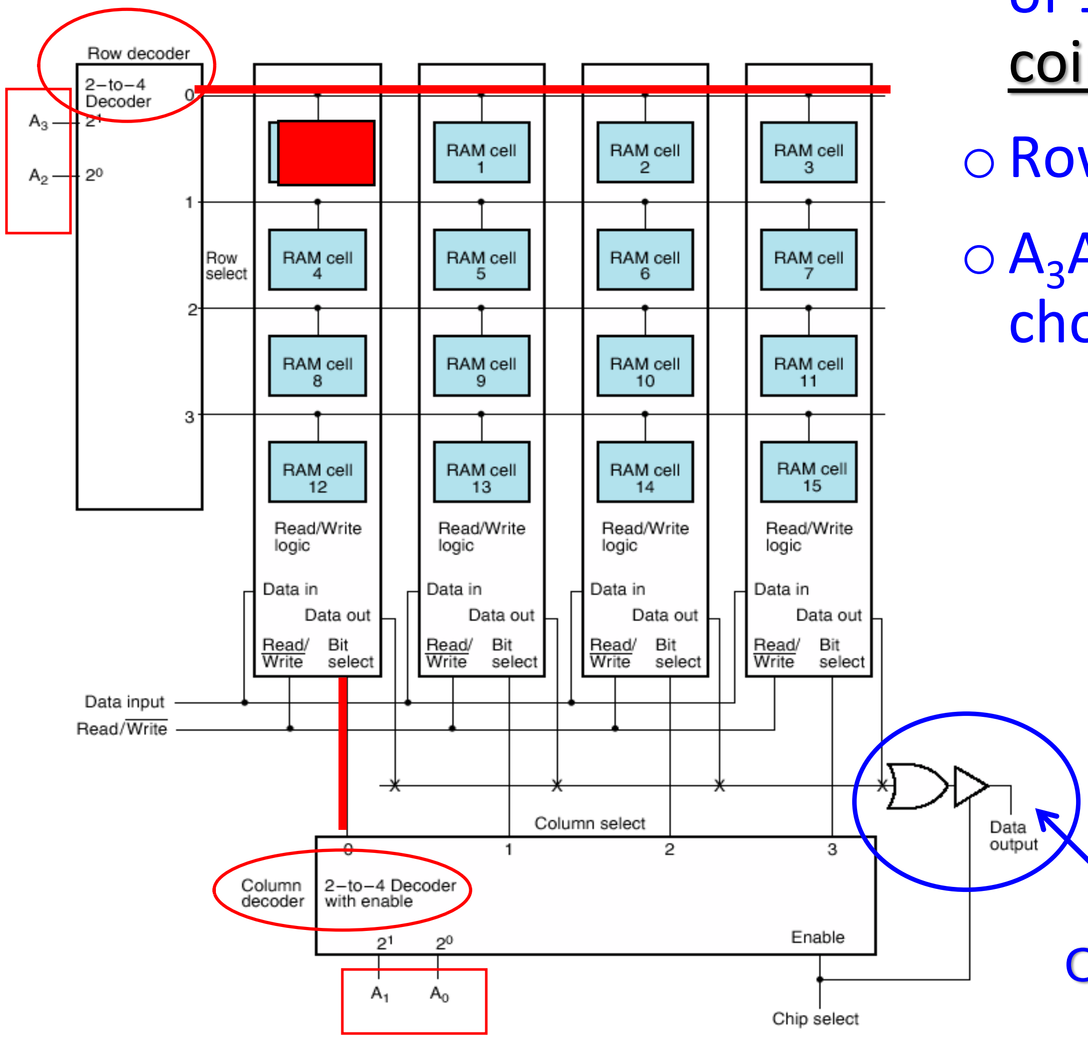
16 X 1 as 4 X 4 Array

Two (2-to-4 line Decoders) → equivalent to 8 2-input AND Gates



- Two decoders
  - Row
  - Column
- Address just broken up
- Not visible from outside

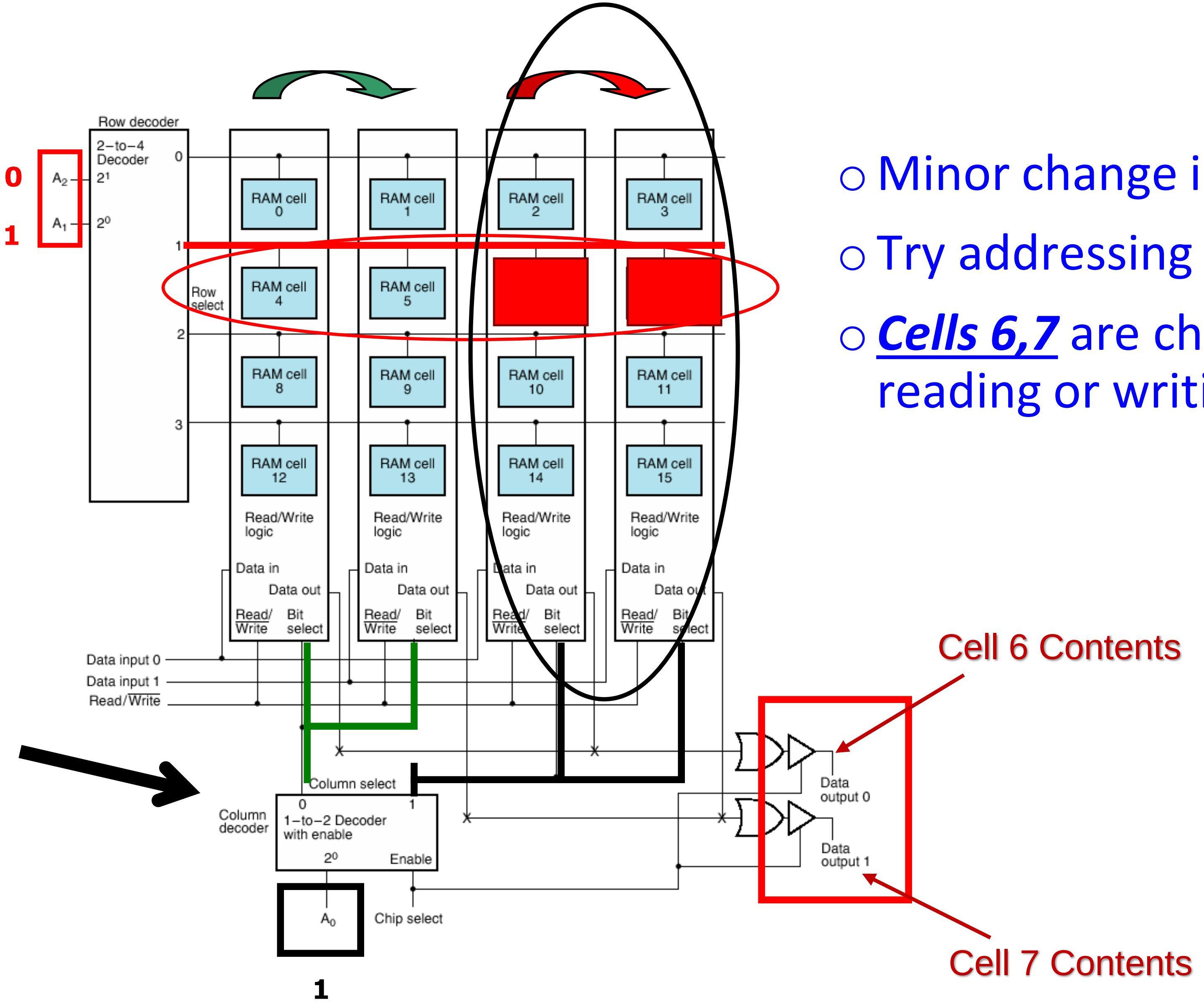
16 X 1 as 4 X 4 Array



- Employing 2 decoders instead of 1 row decoder is called <u>coincident selection</u>
- Row Select and Column Select
- $A_3A_2A_1A_0$=0000 will attempt to choose RAM cell 0.

One bit accessed at a time!

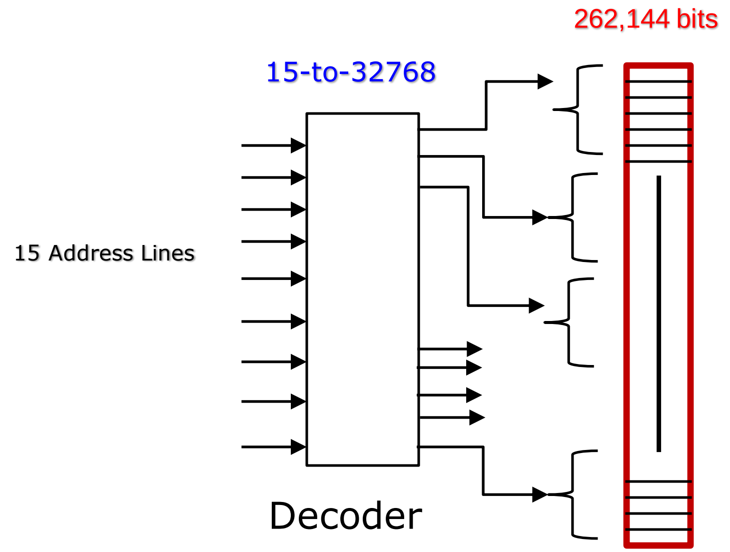
Change from 16x1 to 8 X 2 RAM



- Minor change in logic
- Try addressing 011 on board
- ***Cells 6,7*** are chosen for reading or writing.

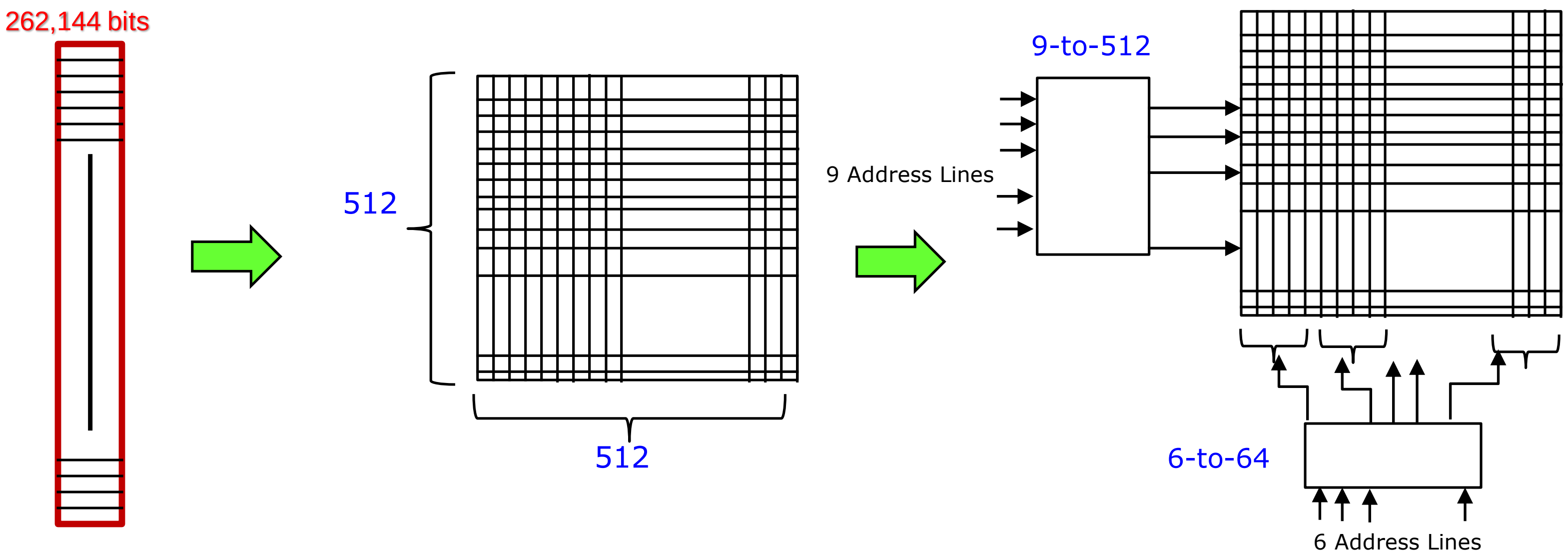
Coincident Selection: Example

- Imagine a 32k x 8 = ((32 x 1024) x 8) = 32,768 x 8
    - 262,144 bits →256K bit memory
- How many address lines required to locate 32K locations (32,768)?
    - Log_2 (32,768) → 15
    - A 15-bit address line is required.
- 262,144/8 = 32,768 chunks each 8-bits
- One column layout would need a decoder with 32,768 outputs
    - ❑ For a single decoder that would mean 32,800 AND gates
    - ❑ This is not practical!
- Solution?
- Coincident selection.



Coincident Selection: Example

- A 32K X 8 contains 256 Kbits (32 x 1024 x 8 = 262,144 bits)
- To make the number of rows and columns equal we take the square root of 256K, giving 512 = 2^9
  - A 9-to-512 decoder is required for the rows (9 address lines are fed to the Row Decoder).
  - Remember we need 8 bits of output!! (Column Decoder?)
- For the columns 512/8 = 64 = 2^6
  - A 6-to-64 decoder is required for the columns (6 address lines are fed to the Column Decoder).
- Total number of gates is now 512+64 = 576 AND gates
- Recall without coincident selection we required a decoder with 32,800 AND gates
- Thus reducing the total gate count by more than 50x.



Error detection and correction

Error Detection and Correction

- Dynamic physical interaction of the electrical signals may cause occasional errors in storing and retrieving
- To improve the reliability, we can use error-detection and error correction codes
- Most common scheme is the parity bit (seen in a previous lecture).
- Memory usually store information in 8, 16 … word size
  - We can use multiple parity bits to check for errors in a word.
  - Parity bits are **calculated** during the **write** process and **checked** during the **read** process
- **Hamming code** is one of the most common error-correction code used in RAMs

Hamming Code

- The most common error-correcting codes used in RAMs was devised by R. W. Hamming
- **k parity** bits are added to an **n-bit data** word
  - **Total bits** (stored) = n+k bits positions numbered from 1 to n+k
  - Position numbers that are power of 2 are reserved for parity bit and the rest is for data
- Example : for 8 bits data (11000100) and 4 bits parity
  - Positions 1, 2, 4, and 8 (all powers of 2) used for parity
  - The rest is used for data
  - note that positions are filled from MSB to LSB

			MSB									LSB
Bit position:	1	2	3	4	5	6	7	8	9	10	11	12
	P_1	P_2	1	P_4	1	0	0	P_8	0	1	0	0

Hamming Code (Calculating the Parity Bits)

- To generate the parity bits P_1-P_4 we use XOR function. How?
  - Which bits are used for each P_x?
- We need to consider the binary representation of each position among n+k positions

Position	Binary representation	Position	Binary representation
1	0001	7	0111
2	0010	8	1000
3	0011	9	1001
4	0100	10	1010
5	0101	11	1011
6	0110	12	1100

- P_1 use all positions with 1 in the least significant bit **1 (1, 3, 5, 7, 9, 11)**
- P_2 use all positions with 1 in the next significant bit **2 (2, 3, 6, 7, 10, 11)**
- P_4 use all positions with 1 in the next significant bit **3 (4, 5, 6, 7, 12)**
- P_8 use all positions with 1 in most significant bit **4 (8, 9, 10, 11,12)**

Hamming Code (Calculating the Parity Bits)

- Now we use the XOR function utilizing the selected positions for each Px
- We are trying to find values for the **positions P_1-P_4 (1,2,4,8)** and they are **not used during the generation** phase of the code
- Recall the bit values/positions

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12
	P_1	P_2	1	P_4	1	0	0	P_8	0	1	0	0
	0	0		1				1				

  - P_1 = XOR of bits (3, 5, 7, 9, 11) = $1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$
  - P_2 = XOR of bits (3, 5, 7, 10, 11) = $1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$
  - P_4 = XOR of bits (5, 6, 7, 12) = $1 \oplus 0 \oplus 0 \oplus 0 = 1$
  - P_8 = XOR of bits (9, 10, 11, 12) = $0 \oplus 1 \oplus 0 \oplus 0 = 1$
- Remember that XOR is odd function and will generate 1 if we have odd number of 1s and 0 otherwise

Our Final code is **0 0 1 1 1 0 0 1 0 1 0 0**

Hamming Code (Error detection)

- When reading the 12 bits from the memory we do the XOR again but now we include the bit values in P_x positions to get for C values
  - C_1 = XOR of bits (1,3, 5, 7, 9, 11)
  - C_2 = XOR of bits (2,3, 5, 7, 10, 11)
  - C_4 = XOR of bits (4, 5, 6, 7, 12)
  - C_8 = XOR of bits (8, 9, 10, 11, 12)
- If the parity code $C = C_8C_4C_2C_1 = 0000$ there is no error
- If $C \neq 0$ the value of C indicates the positions of the bit with error

Hamming Code (Error detection)

- In this example
- First row C = 0000
- Second row C = 0001
- Third row C = 0101
  - Check the values by calculating the XORed result
- The hamming code can detect and correct only a single error
- To detect double errors, you need to add an extra parity bits for the same number of bits in the data

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12	
	0	0	1	1	1	0	0	1	0	1	0	0	No error
	1	0	1	1	1	0	0	1	0	1	0	0	Error in bit 1
	0	0	1	1	0	0	0	1	0	1	0	0	Error in bit 5

Hamming Code (Required number of parity bits)

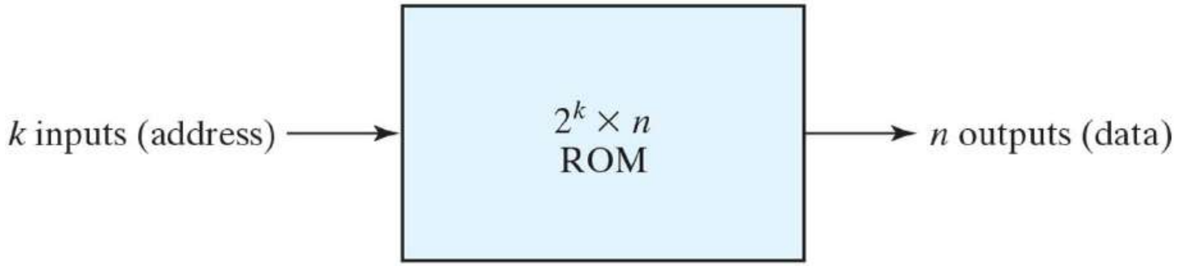
- Hamming can be used for word size larger than 8
- The required number of parity bits k is determined by the word size (range od data bits)

Number of Check Bits, *k*	Range of Data Bits, *n*
3	2–4
4	5–11
5	12–26
6	27–57
7	58–120

Read Only Memory

Read Only Memory

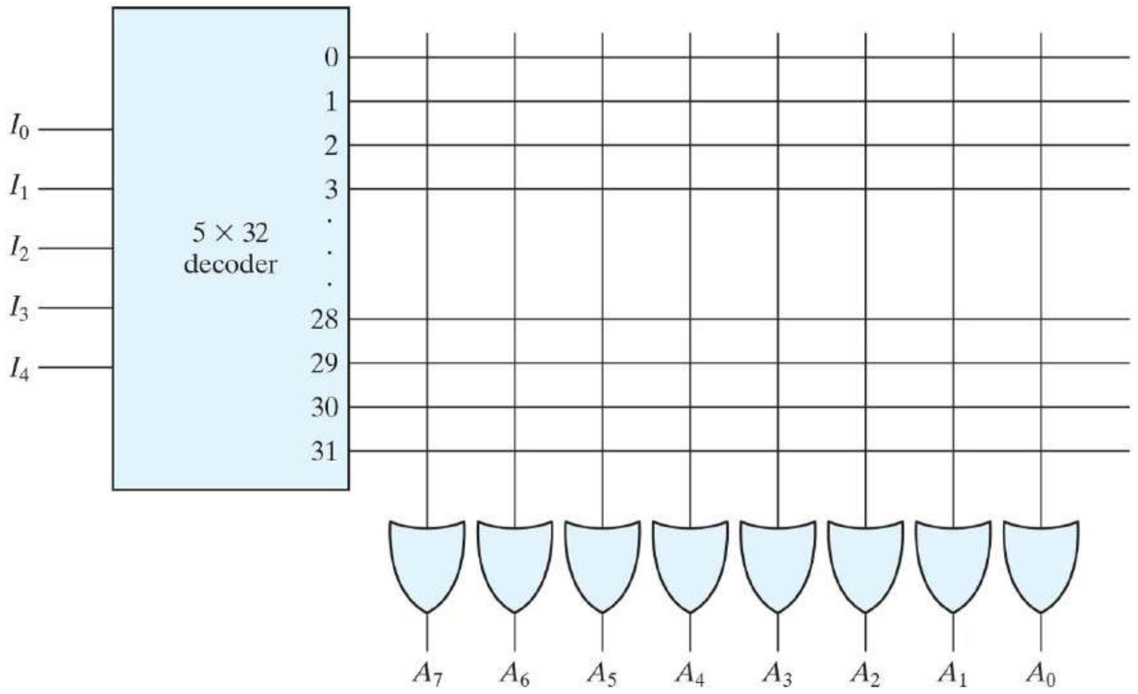
- A read-only memory (ROM) is a memory device in which permanent binary information is stored
- The binary information must be specified by the designer
- Example: A ROM with k inputs and n outputs



- K address can specify one of 2^k words stored in the memory

Read Only Memory

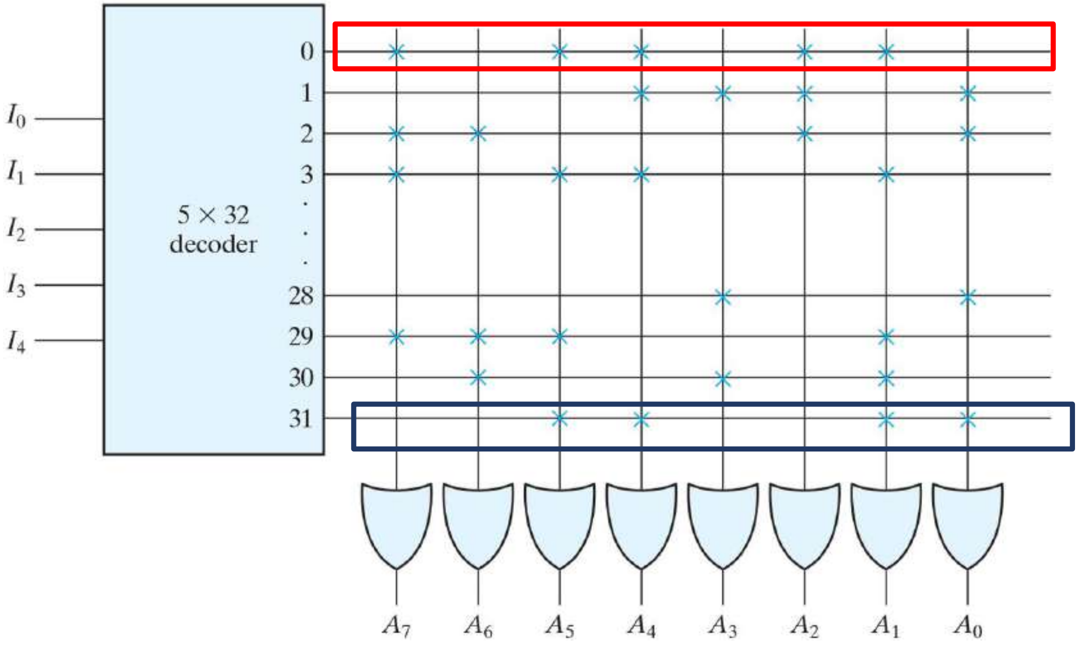
- ROM can be implemented using decoders and OR gates
- Example 32x8 ROM holds 32 word each with 8-bits



Read Only Memory Example

- Data can be stored in the memory by wiring the output of the decoder with different OR gates inputs

Inputs					Outputs							
I_4	I_3	I_2	I_1	I_0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
		⋮						⋮				
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

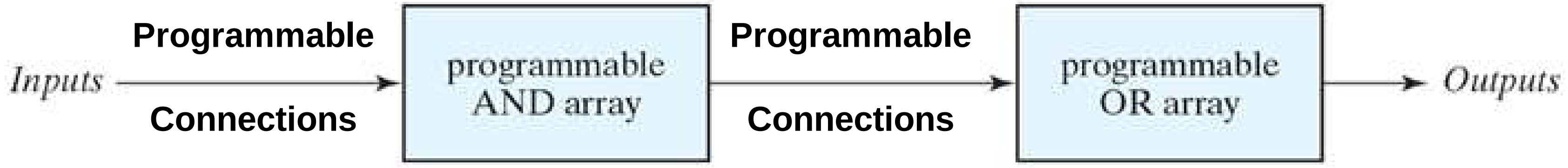


- See example 7.1 in the book

Programable Logic Arrays & FPGAs

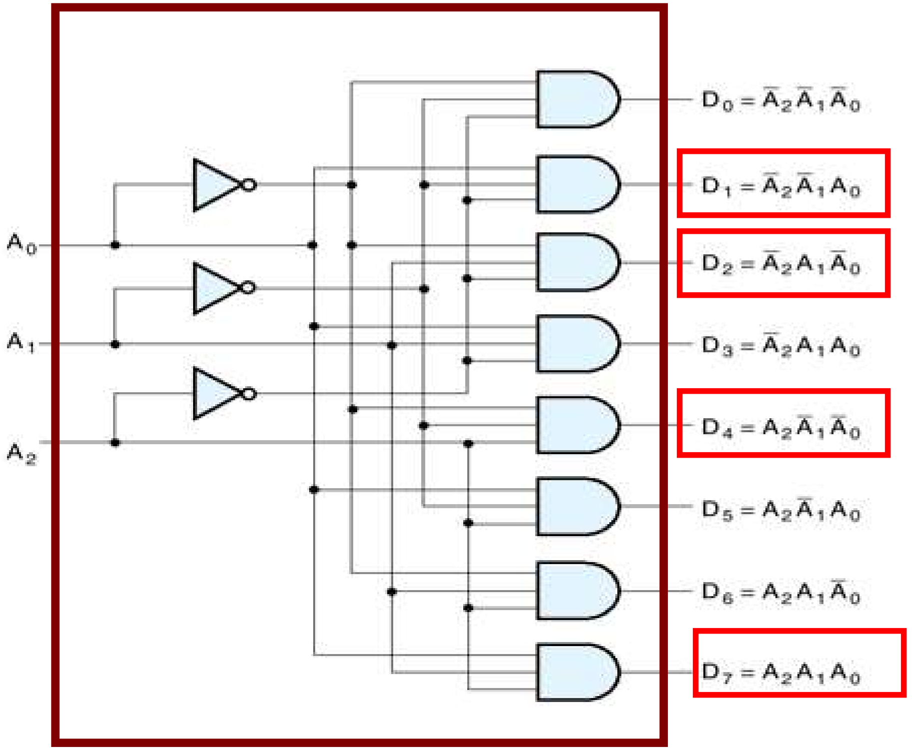
Programmable Logic I

- We learnt in the first part of this course that any combinational logic circuit can be implemented with *sum of min-terms (SOP)*.
- *If we can control* the **inputs** and **number of AND gates** to be used and control the inputs to the OR gate then we can create and design a programmable logic circuit.
- Remember when we used a decoder to implement any Boolean function! That was some type of implementing programmable logic!



Decoders: Implementing Logic

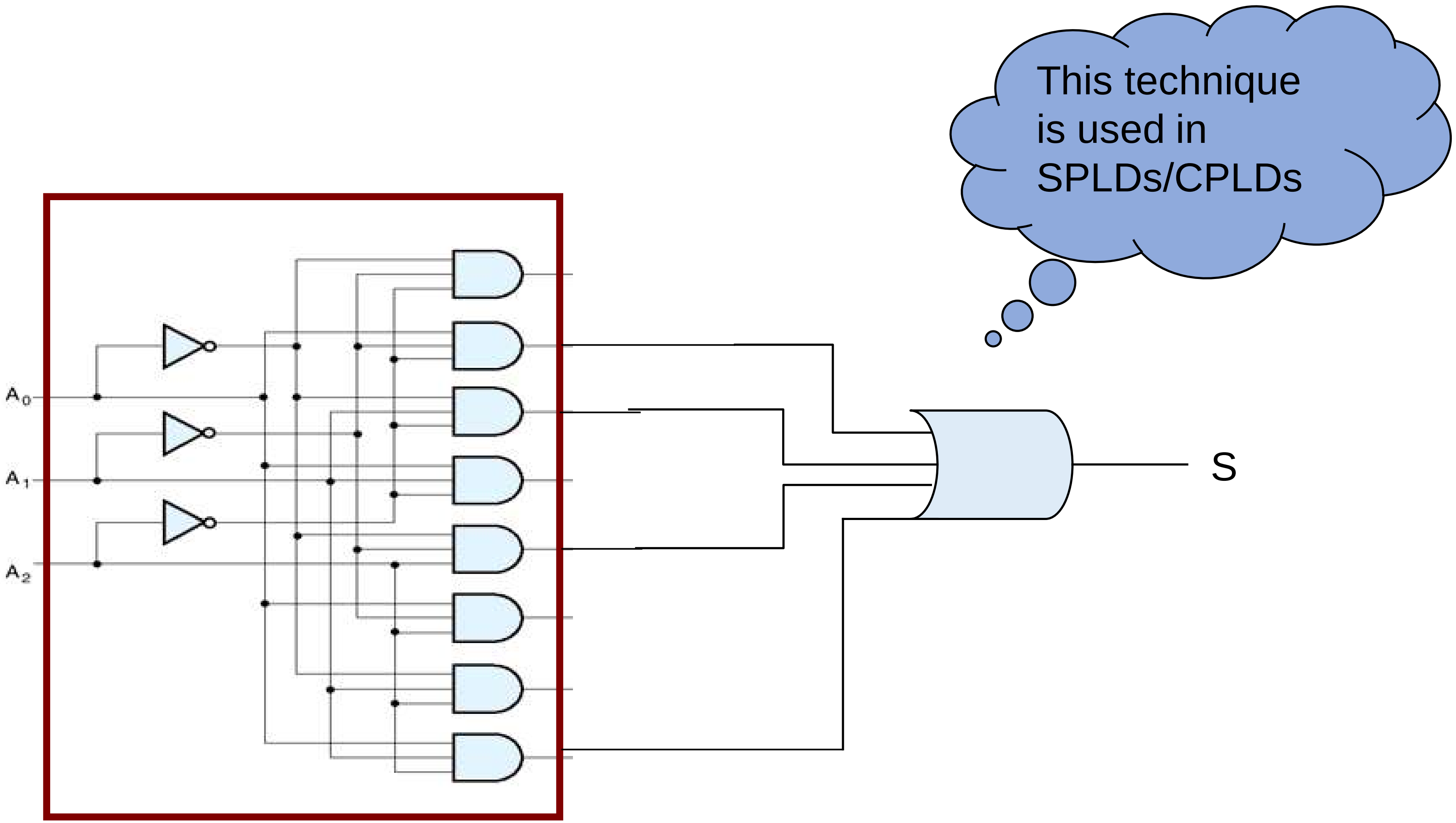
- Example: Implement the following Boolean functions
  1. **$S(A_2,A_1,A_0) = SUM(m(1,2,4,7))$**



1. Since there are <u>three inputs</u>, we need a 3-to-8-line decoder.
2. The decoder generates the <u>eight minterms</u> for inputs A_0,A_1,A_2
3. An OR GATE forms the <u>logical sum</u> minterms required.

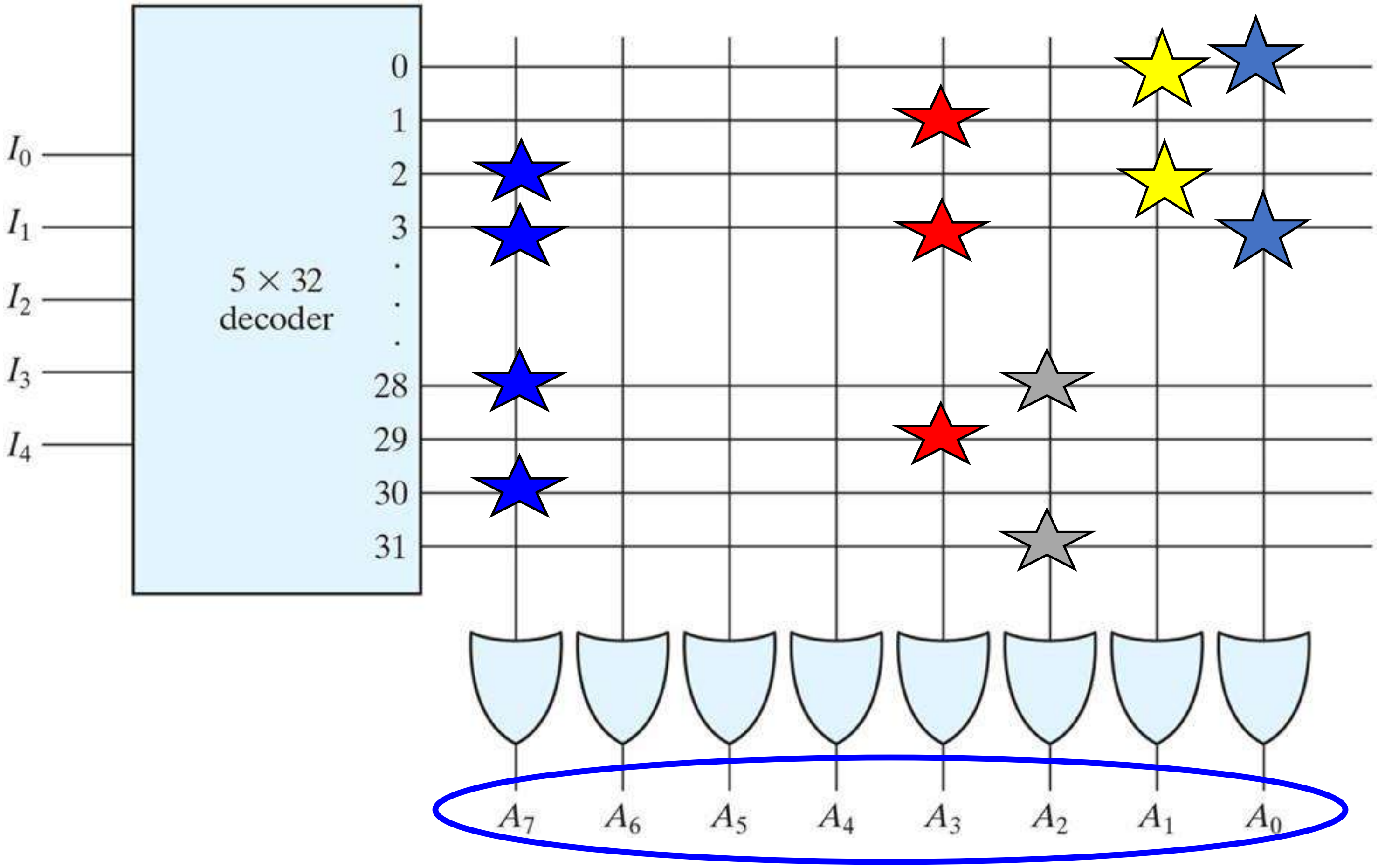
Decoders: Implementing Logic

- Example: Implement the following Boolean functions
  1. $S(A_2,A_1,A_0) = SUM(m(1,2,4,7))$



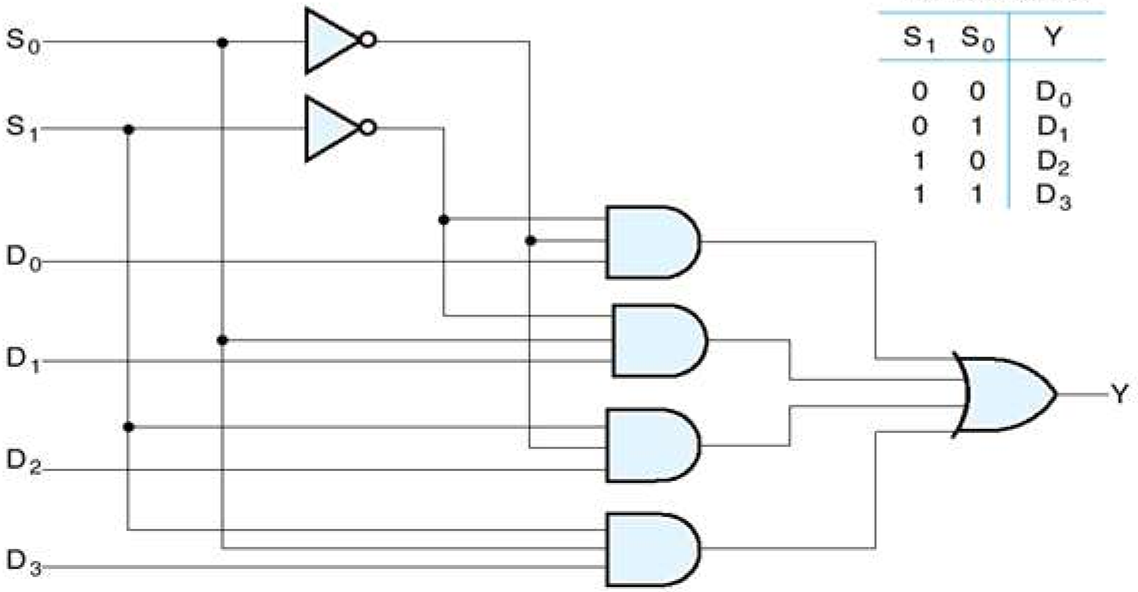
II. Programmable OR Array

We can create 8 different functions with 5-inputs each



Programmable Boolean Functions

Multiplexers can also be used to realize Boolean functions since they consist of an array of AND gates followed by an OR gate.

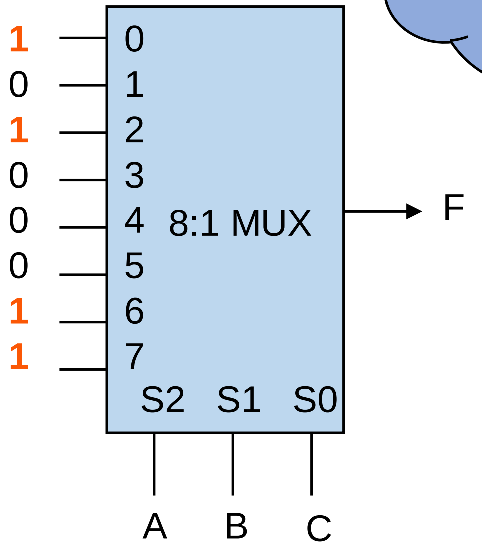


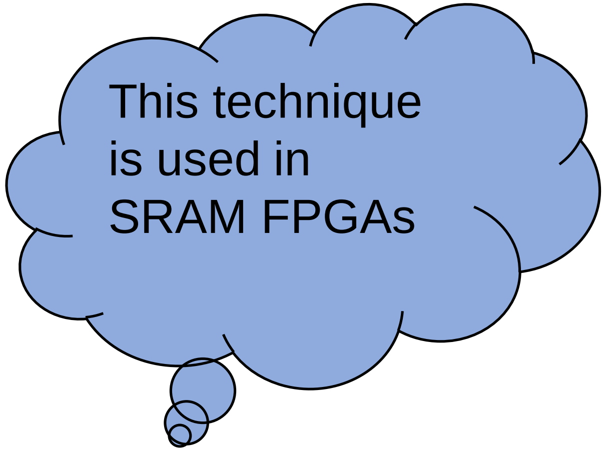
Function table

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

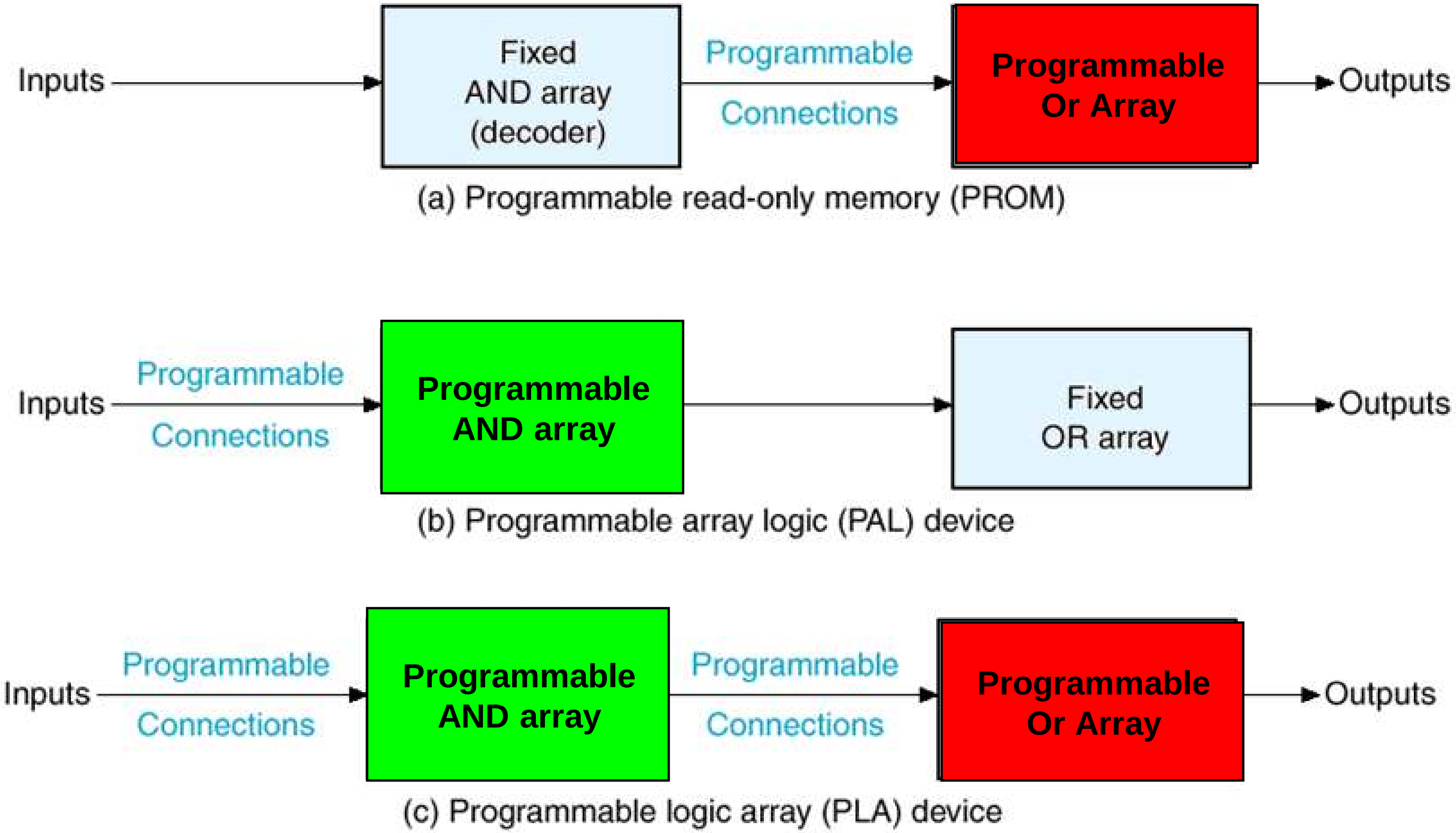
Multiplexers: Implementing Logic

- 2^n:1 multiplexer implements any function of n variables
  1. With the variables used as control inputs and
  2. Data inputs tied to 0 or 1
  3. In essence, *a lookup table*
- Example: F(A,B,C) = m0 + m2 + m6 + m7
  = A'B'C' + A'BC' + ABC' + ABC

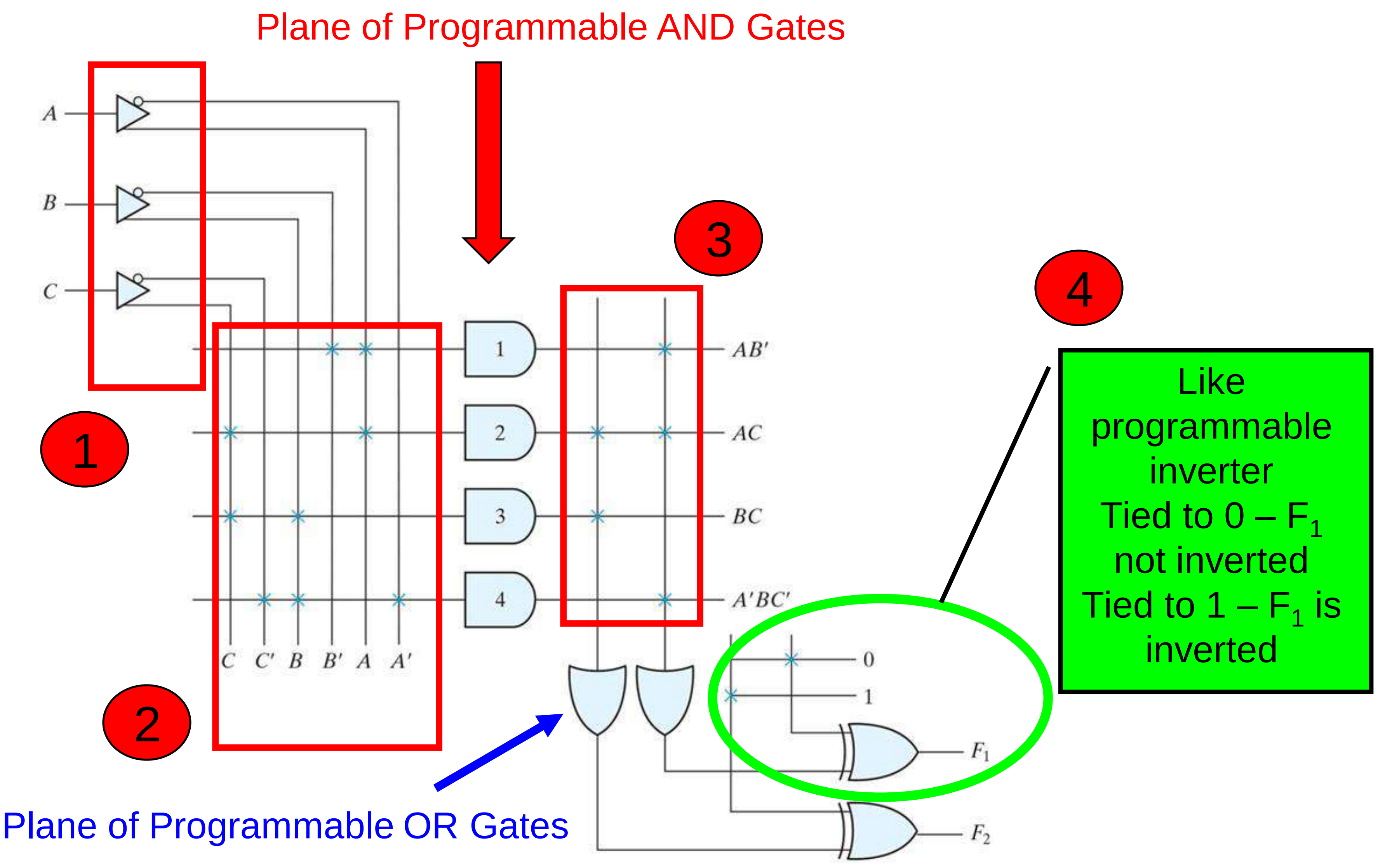




Classification of PLDs



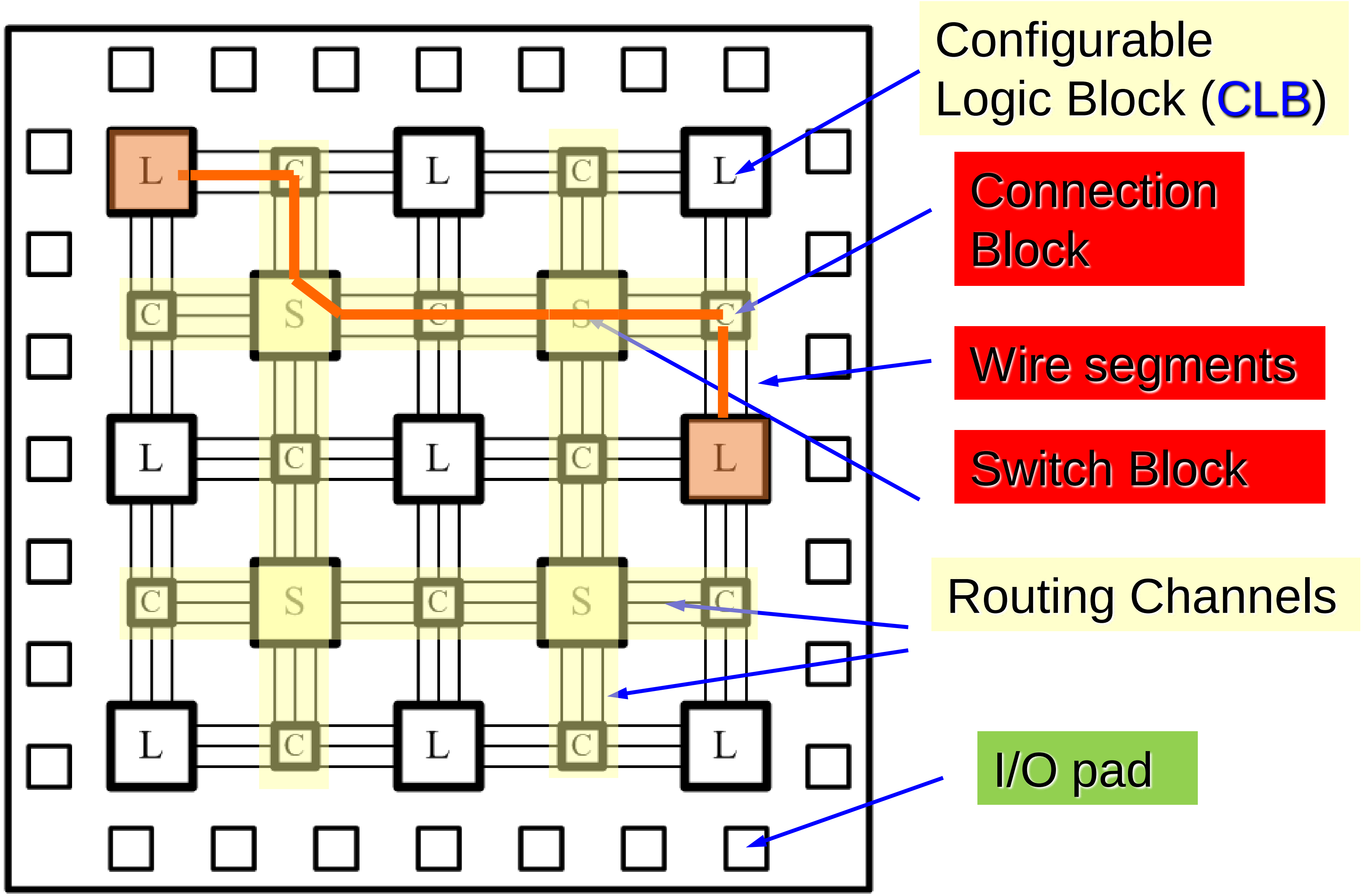
Programmable Logic Array (PLA)



Field-Programmable Gate Array (FPGA)

- FPGAs are circuits that can be programmed by the end user at the user location
- Typical FPGA consists of an array of logic blocks surrounded by programmable input and output blocks together via programmable interconnections
- A typical FPGA logic block consists of lookup tables, multiplexers, gates, and
- flip-flops.

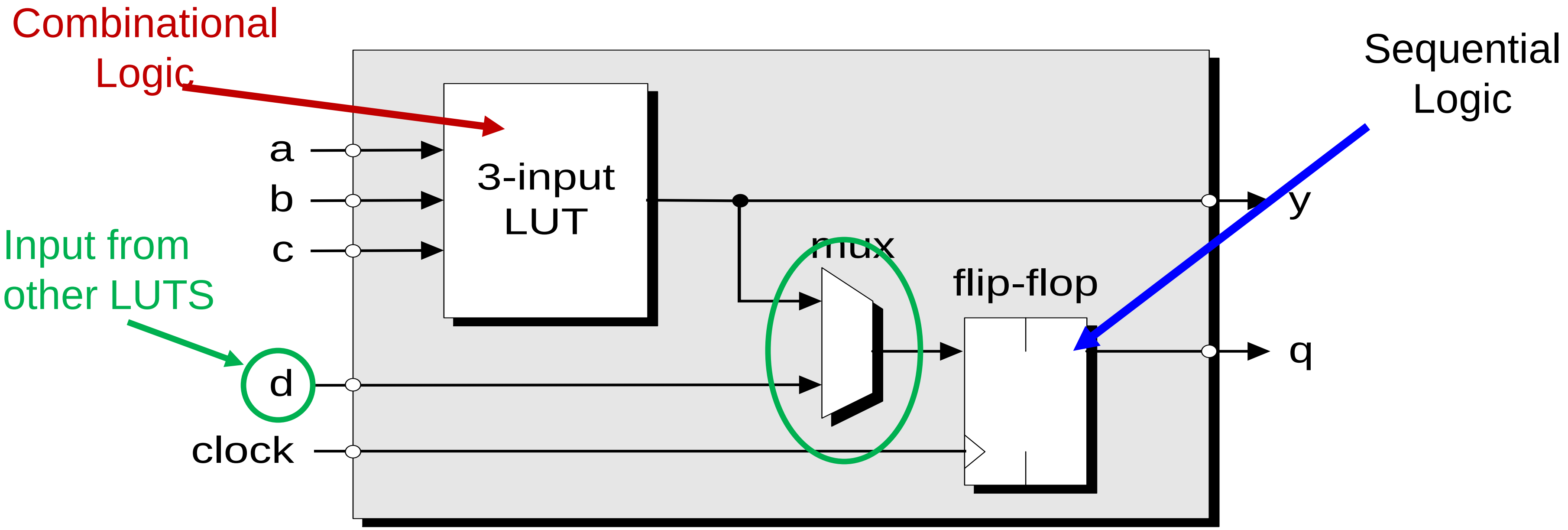
Generic FPGA Architecture:



Programmable Logic Block

Early devices were based on the concept of programmable logic block, which comprised:

1. 3-input lookup table (LUT),
2. register that could act as flip flop or a latch,
3. multiplexer, along with a few other elements.



Basic Logic Element (BLE)

Thank You